Multi-Output GP Emulator Documentation

Release 0.3.1

Eric Daub

Aug 20, 2020

Introduction and Installation:

mogp_emulator is a Python package for fitting Gaussian Process Emulators to computer simulation results. The code contains routines for fitting GP emulators to simulation results with a single or multiple target values, optimizing hyperparameter values, and making predictions on unseen data. The library also implements experimental design, dimension reduction, and calibration tools to enable modellers to understand complex computer simulations.

The following pages give a brief overview of the package, instructions for installation, and an end-to-end tutorial describing a Uncertainty Quantification workflow using mogp_emulator. Further pages outline some additional examples, more background details on the methods in the MUCM Toolkit, full implementation details, and some included benchmarks.

CHAPTER 1

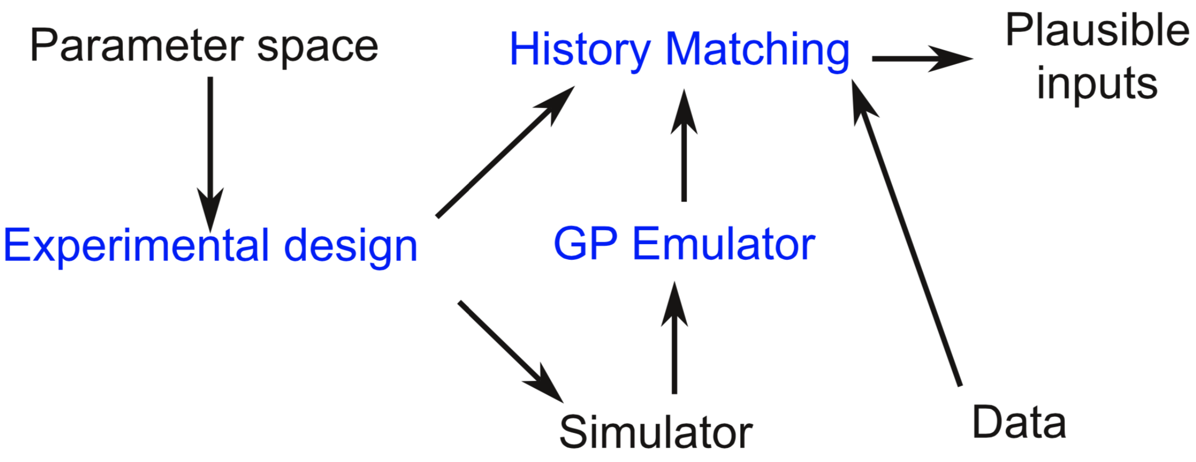
Overview

Computer simulations are frequently used in science to understand physical systems, but in practice it can be difficult to understand their limitations and quantify the uncertainty associated with their outputs. This library provides a range of tools to facilitate this process for domain experts that are not necessarily well-versed in uncertainty quantification methods.

This page covers an overview of the workflow. For much more detail, see the *Details on the Method* section of the introduction, or the *Uncertainty Quantification Methods* section.

1.1 UQ Basics

The UQ workflow here describes the process of understanding a complex simulator. The simulator here is assumed to be a deterministic function mapping multiple inputs to one or more outputs with some general knowledge about the input space. We would like to understand what inputs are reasonable given some observations about the world, and understand how uncertain we are about this knowledge. The inputs may not be something that is physically observable, as they may be standing in for missing physics from the simulator.



The simulation is assumed to be too expensive

Fig. 1: The Uncertainty Quantification workflow. To determine the plausible inputs for a complex simulator given some data and a parameter space, we use an **experimental design** to sample from the space, run those points through the simulator. To approximate the simulator, we fit a **Gaussian Process Emulator** to the simulator output. Then, to explore the parameter space, we sample many, many times from the experimental design, query the emulator, and determine the plausible inputs that are consistent with the data using **History Matching**.

to run as many times as needed to explore the entire space. Instead, we will train a **surrogate model** or **emulator** model that approximates the simulator and efficiently estimates its value (plus an uncertainty). We will then query that model many times to explore the input space.

Because the simulator is expensive to run, we would like to be judicious about how to sample from the space. This requires designing an experiment to choose the points that are run to fit the surrogate model, referred to as an **Experimental Design**. Once these points are run, the emulator can be fit and predictions made on arbitrary input points. These predictions are compared to observations for a large number of points to examine the inputs space and see what inputs are reasonable given the observations and what points can be excluded. This is a type of model calibration, and the specific approach we use here is **History Matching**, which attempts to find the parts of the input space that are plausible given the data and all uncertainties involved in the problem.

Next, we describe the *Installation* prodecure and provide an example *Tutorial* to illustrate how this process works.

CHAPTER 2

Installation

In most cases, the easiest way to install `mogp_emulator` is via `pip`, which should install the library and all of its dependencies:

```
pip install mogp_emulator
```

(NOTE: the project is not yet on PyPI, but we expect that the current version of the `devel` branch will be the first one available to be installed that way so are accounting for this in the rewrite of this documentation. Until that is completed, please follow the instructions below to install the software manually. Thank you for your patience as we are hard at work on improvements to the software!)

2.1 Manual Installation

To install the package manually, for instance to have access to a development version or to take part in active development of the package, the following instructions can be used to install the package.

2.1.1 Download

You can download the code as a zipped archive from the Github repository. This will download all files on the *master* branch, which can be unpacked and then used to install following the instructions below.

If you prefer to check out the Github repository, you can download the code using:

```
git clone https://github.com/alan-turing-institute/mogp_emulator/
```

This will clone the entire git history of the software and check out the `master` branch by default. The `master` branch is the most stable version of the code, but will not have all features as the code is under active development. The `devel` branch is the more actively developed branch, and all new features will be available here as they are completed. All code in the `devel` branch is well tested and documented to the point that results can be trusted, but may still have some minor bugs and issues. Any other branch is used to develop new features and should be considered untested and experimental. Please get in touch with one of the team members if you are unsure if a particular feature is available.

2.1.2 Requirements

The code requires Python 3.6 or later, and working Numpy and Scipy installations are required. You should be able to install these packages using pip if you do not have them already available on your system. From the base mogp_emulator directory, you can install all required packages using:

```
pip install -r requirements.txt
```

This will install the minimum requirements needed to use mogp_emulator. There are a few addditional packages that are not required but can be useful (in particular, patsy is used for parsing mean functions using R-style formulas, so all R users are highly encouraged to install the optional dependencies). Installation of the optional dependencies can be done via:

```
pip install -r requirements-optional.txt
```

2.1.3 Installation

Then to install the main code, run the following command:

```
python setup.py install
```

This will install the main code in the system Python installation. You may need adminstrative priveleges to install the software itself, depending on your system configuration. However, any updates to the code cloned through the github repository (particularly if you are using the devel branch, which is under more active development) will not be reflected in the system installation using this method. If you would like to always have the most active development version, install using:

```
python setup.py develop
```

This will insert symlinks to the repository files into the system Python installation so that files are updated whenever there are changes to the code files.

2.2 Documentation

The code documentation is available on readthedocs. A current build of the master and devel branches should always be available in HTML or PDF format.

To build the documentation yourself requires Sphinx, which can be installed using pip. This can also be done in the docs directory using pip install -r requirements.txt. To build the documentatation, change to the docs directory. There is a Makefile in the *docs* directory to facilitate building the documentation for you. To build the HTML version, enter the following into the shell from the docs directory:

```
make html
```

This will build the HTML version of the documentation. A standalone PDF version can be built, which requires a standard LaTeX installation, via:

```
make latexpdf
```

In both cases, the documentation files can be found in the corresponding directories in the docs/_build directory. Note that if these directories are not found on your system, you may need to create them in order for the build to finish correctly. A version of the documentation can also be found at the link above on Read the Docs.

2.3 Testing the Installation

2.3.1 Unit Tests

mogp_emulator includes a full set of unit tests. To run the test suite, you will need to install the development dependencies, which include pytest and pytest-cov to give coverage reports, which can be done in the main mogp_emulator directory via pip install -r requirements-dev.txt. The pytest-cov package is not required to run the test suite, but is useful if you are developing the code to determine test coverage.

The tests can be run from the base mogp_emulator directory or the mogp_emulator/tests directory by entering pytest, which will run all tests and print out the results to the console. In the mogp_emulator/tests directory, there is also a Makefile that will run the tests for you. You can simply enter make tests into the shell.

CHAPTER 3

Tutorial

Note: This tutorial requires Scipy version 1.4 or later to run the simulator.

This page includes an end-to-end example of using `mogp_emulator` to perform model calibration. We define a simulator describing projectile motion with nonlinear drag, and then illustrate how to sample from the simulator, fit a surrogate model, and explore the parameter space using history matching to obtain a plausible subset of the input space.

3.1 Projectile Motion with Drag

In this example, we will explore projectile motion with drag. A projectile with a mass of 1 kg is fired at an angle of 45 degrees from horizontal and a height of 2 meters. The projectile experiences gravity and drag opposes the motion of the projectile, with a force that is proportional to the squared velocity. For this example, we assume that we do not know the initial velocity or the drag coefficient, but we do know that the projectile travelled 2 km (with a standard deviation measurement error of 20 meters). We would like to determine what inputs could have resulted in this observation.

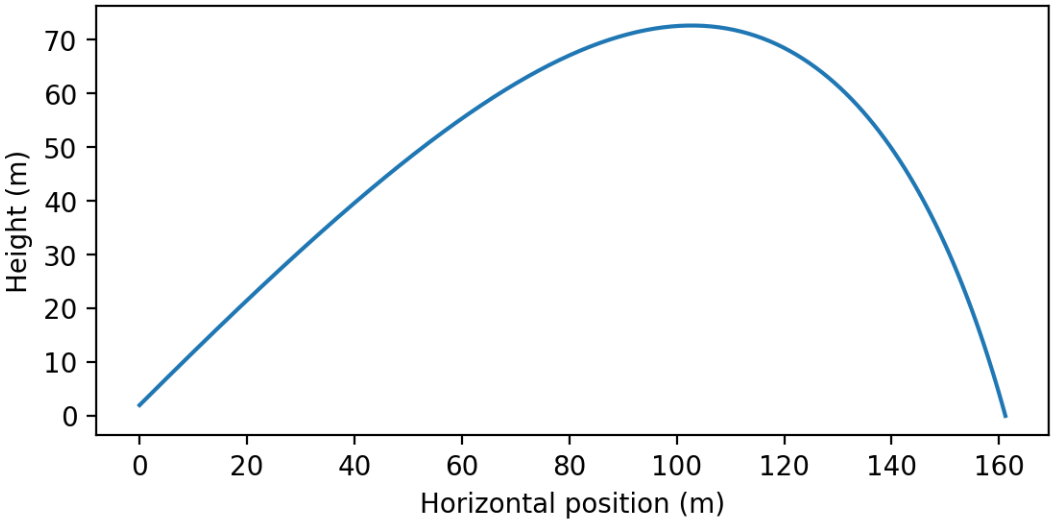


Fig. 1: Example simulation of projectile motion with drag for $C = 0.01$ kg/m and $v_0 = 100$ m/s computed via numerical integration.

3.1.1 Equations of Motion

The equations of motion for the projectile are as follows:

$$
\begin{aligned}
m\frac{dv_x}{dt} &= -Cv_x\sqrt{v_x^2+v_y^2} \\
m\frac{dv_y}{dt} &= -g - Cv_y\sqrt{v_x^2+v_y^2} \\
\frac{dx}{dt} &= v_x \\
\frac{dy}{dt} &= v_y
\end{aligned}
$$

The initial conditions are

$$
\begin{aligned}
v_{x0} &= v_0/\sqrt{2} \\
v_{y0} &= v_0/\sqrt{2} \\
x_0 &= 0 \\
y_0 &= h
\end{aligned}
$$

This can be integrated in time until $y = 0$, and the distance travelled is the value of x when this occurs.

We include a Python implementation of this model in `mogp_emulator/demos/projectile.py`. This uses the `scipy.integrate` function `solve_ivp` to perform the numerical integration. The RHS derivative is defined in `f`, and the stopping condition is defined in the `event` function. The simulator is then defined as a function taking a single input (an array holding the two input parameters of the drag coefficient C and the initial velocity v_0) and returning a single value, which is x at the end of the simulation.

```
import numpy as np
import scipy
from scipy.integrate import solve_ivp

# define functions needed for simulator

def f(t, y, c):
    "Compute RHS of system of differential equations, returning vector derivative"

    # check inputs and extract

    assert len(y) == 4
    assert c >= 0.

    vx = y[0]
    vy = y[1]

    # calculate derivatives

    dydt = np.zeros(4)

    dydt[0] = -c*vx*np.sqrt(vx**2 + vy**2)
    dydt[1] = -9.8 - c*vy*np.sqrt(vx**2 + vy**2)
    dydt[2] = vx
    dydt[3] = vy
```

(continues on next page)

(continued from previous page)

```
    return dydt

def event(t, y, c):
    "event to trigger end of integration"

    assert len(y) == 4
    assert c >= 0.

    return y[3]

# now can define simulator

def simulator(x):
    "simulator to solve ODE system for projectile motion with drag. returns distance␣
↪projectile travels"

    # unpack values

    assert len(x) == 2
    assert x[1] > 0.

    c = 10.**x[0]
    v0 = x[1]

    # set initial conditions

    y0 = np.zeros(4)

    y0[0] = v0/np.sqrt(2.)
    y0[1] = v0/np.sqrt(2.)
    y0[3] = 2.

    # run simulation

    results = solve_ivp(f, (0., 1.e8), y0, events=event, args = (c,))

    return results.y_events[0][0][2]

# function for printing out results

def print_results(inputs, predictions):
    "convenience function for printing out results and computing mean square error"

    print("Target Point                   Predicted mean            Actual Value")
    print("---------------------------------------------------------------------------
↪---")

    error = 0.

    for pp, m in zip(inputs, predictions):
        trueval = simulator(pp)
        print("{}      {}       {}".format(pp, m, simulator(pp)))
        error += (trueval - m)**2

    print("Mean squared error: {}".format(np.sqrt(error)/len(predictions)))
```

3.1.2 Parameter Space

We are not sure what values of the parameters to use, so we must pick reasonable ranges. The velocity range might be somewhere in the range of 0-1000 m/s, while the drag coefficient is much more uncertain. For this reason, we use a logarithmic scale to represent the drag coeffient, with values ranging from 10^{-5} to 10 kg/m. This will ensure that we sample from a wide range of values to ensure that we understand the effect of this parameter on the simulation.

3.2 UQ Implementation

As described in *Overview*, Our analysis consists of three steps:

1. Drawing parameter values to run our simulator
2. Fitting a surrogate model to those points
3. Performing model calibration by sampling many points and comparing to the observations

We will describe each step and provide some code illustrating how the steps are done in `mogp_emulator` below. The full example is provided in the file `mogp_emulator/demos/tutorial.py`, and we provide snippets here to illustrate.

3.2.1 Experimental Design

For this example, we use a *Latin Hypercube Design* to sample from the parameter space. Latin Hypercubes attempt to draw from all parts of the distribution, and for small numbers of samples are likely to outperform Monte Carlo sampling.

To define a Latin Hypercube, we must give it the base distributions for all input parameters from which to draw the samples. Because we would like our drag coefficient to be uniformly distributed on a log scale, and the initial velocity to be uniformly distributed on a linear scale, we simply need to provide the upper and lower bounds of the uniform distribution and the Python object will create the distributions for us. If we wanted to use a more complicated distribution, we can pass `scipy.stats` Point Probability Functions (the inverse of the CDF) when constructing the *LatinHypercubeDesign* object instead. However, in practice we often do not know much about the parameter distributions, so uniform distributions are fairly common due to their simplicity.

To construct our experimental design and draw samples from it, we do the following:

```
import numpy as np
from projectile import simulator, print_results
import mogp_emulator

lhd = mogp_emulator.LatinHypercubeDesign([(-5., 1.), (0., 1000.)])

n_simulations = 50
simulation_points = lhd.sample(n_simulations)
simulation_output = np.array([simulator(p) for p in simulation_points])
```

This constructs an instance of *LatinHypercubeDesign*, and creates the underlying distributions by providing a list of tuples. Each tuple gives the upper and lower bounds on the uniform distribution. Thus, the first tuple determines the drag coefficient (recall that it is on a log scale, so this is defining the distribution on the exponent), and the second determines the initial velocity.

Next, we determine that we want to run 50 simulations. We can get our simulation points by calling the `sample` method of *LatinHypercubeDesign*, which is a numpy array of shape `(n_simulations, 2)`. Thus, iterating over the resulting object gives us the parameters for each of our simulations.

We can then simply run our simulation in our Python script. However, for more complicated simulations, we may need to save these values and then submit our jobs to a computer cluster to have the simulations run in a reasonable amount of time.

3.2.2 Gaussian Process Emulator

Once we have our simulation points, we fit our surrogate model using the *GaussianProcess* class. Fitting this model involves giving the GP object our inputs and our targets, and then fitting the parameters of the model using an estimation technique such as Maximum Likelihood Estimation. This is done by passing the GP object to the `fit_GP_MAP` function, which returns the same GP object but with the parameter values estimated.

```
gp = mogp_emulator.GaussianProcess(simulation_points, simulation_output)
gp = mogp_emulator.fit_GP_MAP(gp)

print("Correlation lengths = {}".format(np.sqrt(np.exp(-gp.theta[:2]))))
print("Sigma = {}".format(np.sqrt(np.exp(gp.theta[2]))))
```

While the function is called `fit_GP_MAP` (MAP means Maximum A Posteriori), in this case we have not provided any prior information on the parameter values, so it results in MLE.

Following fitting, we print out some of the hyperparameters that are estimated. First, we print out the correlation lengths estimated for each of the input parameters. These determine how far we have to move in that coordinate direction to see a significant change in the output. If you run this example, if you get a decent fit you should see correlation lengths of ~ 1.3 and ~ 500 (your values may differ a bit, but note that the fit is not highly sensitive to these values). The overall variation in the function is captured by the variance scale σ, which should be around $\sim 20,000$ for this example.

If your values are very different from these, there is a good chance your fit is not very good (perhaps due to poor sampling). If that is the case, you can run the script again until you get a reasonable fit.

3.2.3 Emulator Validation

To show that the emulator is doing a reasonable job, we now cross validate the emulator to compare its predictions with the output from the simulator. This involves drawing additional samples and running the simulations as was done above. However, we also need to predict what the GP thinks the function values are and the uncertainty. This is done with the `predict` method of *GaussianProcess*:

```
n_valid = 10
validation_points = lhd.sample(n_valid)
validation_output = np.array([simulator(p) for p in validation_points])

predictions = gp.predict(validation_points)

print_results(validation_points, predictions.mean)

# Finally, perform history matching. Sample densely from the experimental design and
```

`predictions` is an object containing the mean and uncertainty (variance) of the predictions. A GP assumes that the outputs follow a Normal Distribution, so we can perform validation by asking how many of our validation points mean estimates are within 2 standard deviations of the true value. Usually for this example this is about 8/10, so not quite as we would expect if it were perfectly recreating the function. However, we will see that this still is good enough in most cases for the task at hand.

3.2.4 History Matching

The final step in the analysis is to perform calibration, where we draw a large number of samples from the model input and compare the output of the surrogate model to the observations to determine what inputs are plausible given the data. There are many ways to perform model calibration, but we think that History Matching is a robust technique well-suited for most problems. It has the particular advantage in that even in the situation where the surrogate model is not particularly accurate, the results from History Matching are still valid. This is in contrast to full Bayesian Calibration, where the surrogate model must be accurate over the entire input space to obtain good results.

History matching involves computing an **implausibility metric**, which determines how likely a particular set of inputs describes the given observations. There are many choices for how to compute this metric, but we default to the simplest version where we compute the number of standard deviations between the surrogate model mean and the observations. The variance is determined by summing the observation error, the surrogate model error, and a final error known as **model discrepancy**. Model discrepancy is meant to account for the fact that our simulations do not completely describe reality, and is an important consideration in studying most complex physical models. In this example, however, we assume that our model is perfect and the model discrepancy is zero, though we will still consider the other two sources of error.

To compute the implausibility metric, we need to draw a much larger number of samples from the experimental design to ensure that we have good coverage of the input parameter space (it is not uncommon to make millions of predictions when doing history matching in research problems). We draw from our Latin Hypercube Design again, though at this sampling density there is probably not a significant difference between the Latin Hypercube and Monte Carlo sampling (especially in only 2 dimensions). Then, we create a *HistoryMatching* object and compute which points are “Not Ruled Out Yet” (NROY). This is done as follows:

```

hm = mogp_emulator.HistoryMatching(gp=gp, coords=prediction_points, obs=[2000., 400.])

nroy_points = hm.get_NROY()

print("Ruled out {} of {} points".format(n_predict - len(nroy_points), n_predict))

# If plotting enabled, visualize results
```

First, we set a large number of samples and draw them from the experimental design object. Then, We construct the *HistoryMatching* object by giving the fit GP surrogate model (the `gp` argument), the prediction points to consider (the `coords` argument), and the observations (the `obs` argument) as an observed value with an uncertainty (as a variance). The `predict` method of the GP object is used to make predictions inside the history matching class. With the constructed *HistoryMatching* object, we can obtain the NROY points by calling the `get_NROY` method. This returns a list of integer indices that can be used to index into the `prediction_points` array and learn about the points that are not ruled out by our analysis. We finally print out the fraction of points that *were* ruled out. In most cases, this should be a large fraction of the space, usually around 98% of the sampled points. Those that are *not* ruled out are plausible inputs given the data.

We can visualize this quite easily due to the fact that our parameter space is only 2D by making a scatter plot of the NROY points. We also include the sample points used to construct the surrogate model for reference. This plotting command is only executed if `matplotlib` is installed:

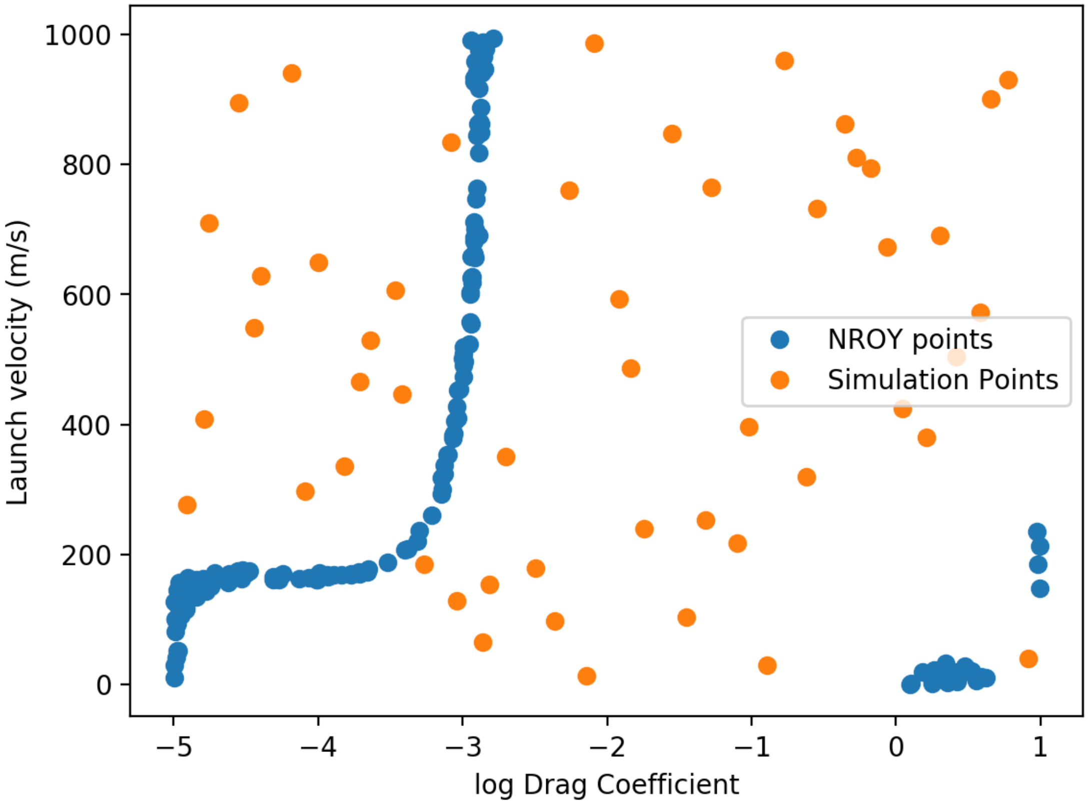
```
try:
    import matplotlib.pyplot as plt
    makeplots = True
except ImportError:
    makeplots = False

    plt.plot(simulation_points[:,0], simulation_points[:,1],"o", label="Simulation␣
↪Points")
```

(continues on next page)

(continued from previous page)

```
    plt.xlabel("log Drag Coefficient")
    plt.ylabel("Launch velocity (m/s)")
    plt.legend()
    plt.show()
```

which should make a plot that looks something like this:



If the original emulator makes accurate predictions, you should get something that looks similar to the above plot. As you can see, most of the space can be ruled out, and only a small fraction of the points remain as plausible options. For launch velocities below around 200 m/s the projectile cannot reach the observed distance regardless of the drag coefficient. Above this value, a narrow range of (C, v_0) pairs are allowed (presumably a line plus some error due to the observation error if our emulator could exactly reproduce the simulator solution). Above a drag coefficient of around 10^{-3} kg/m, none of the launch velocities that we sampled can produce the observations as the drag is presumably too high for the projectile to travel that distance. There are some points at the edges of the simulation that we cannot rule out, though the fact that they occur in gaps in the input simulation sampling suggests that they are likely due to errors in our emulator in those regions.

3.3 More Details

This simple analysis illustrates the basic approach to running a model calibration example. In practice, this simulator is not particularly expensive to run, and so we could imagine doing this analysis without the surrogate model. However, if the simulation takes even 1 second, drawing the 10,000 samples needed to explore the parameter space would take 3 hours, and a million samples would take nearly 2 weeks. Thus, the surrogate becomes necessary very quickly if we wish to exhaustively explore the input space to the point of being confident in our sampling.

More details about these steps can be found in the *Uncertainty Quantification Methods* section, or on the following page that goes into *more details* on the options available in this software library. For more on the specific implementation detials, see the various *implementation pages* describing the software components.

CHAPTER 4

Details on the Method

The UQ workflow described in the *Overview* section has three main components: Experimental Design, a Gaussian Process Emulator, and History Matching, each of which we describe in more detail on this page. For more specifics on the software implementation, see the linked pages to the individual classes provided with this library.

4.1 Experimental Design

To run the simulator, we must first select the inputs given some broad knowledge of the parameter space. This is done using the various *ExperimentalDesign* classes, which require that the user specifies the distribution from which parameter values are drawn. Depending on the particular design used, the design computes a desired number of points to sample for the experimental design.

The simplest approach is to use a *Monte Carlo Design*, which simply randomly draws points from the underlying distributions. However, in practice this does not usually give the best performance, as no attempt is made to draw points from the full range of the distribution. To improve upon this, we can use a *Latin Hypercube Design*, which guarantees that the every sample is drawn from a different quantile of the underlying distribution.

In practice, however, because the computation required to fit a surrogate model is usually small when compared to the computational effort in running the simulator, a *Sequential Design* can provide improved performance. A sequential design is more intelligent about the next point to sample by determining what new point from a set of options will improve the surrogate model. This package provides an implementation of the *Mutual Information for Computer Experiments (MICE)* sequential design algorithm, which has been shown to outperform Latin Hypercubes with only a small additional computational overhead.

4.2 Gaussian Process Emulator

The central component of the UQ method is *Gaussian Process* regression, which serves as the surrogate model in the UQ workflow adopted here. Given a set of input variables and target values, the Gaussian Process interpolates those values using a multivariate Gaussian distribution using user-specified mean and covariance functions and priors on the hyperparameter values (if desired). Fitting the Gaussian process requires inverting the covariance matrix computed from the training data, which is done using Cholesky decomposition as the covariance matrix is symmetric and positive definite (complexity is $\mathcal{O}(n^3)$, where n is the number of training points). The squared exponential covariance function contains several hyperparameters, which includes a length scale for each input variable and an overall variance. These hyperparameters can be set manually, or chosen automatically by minimizing the negative logarithm of the posterior marginalized over the data. Once the hyperparameters are fit, predictions can be made efficiently (complexity $\mathcal{O}(n)$ for each prediction, where n is again the number of training points), and the variance computed (complexity $\mathcal{O}(n^2)$ for each prediction).

The code assumes that the simulations are exact and attempts to interpolate between them. However, in some cases, if two training points are too close to one another the resulting covariance matrix is singular due to the co-linear points. A "nugget" term (noise added to the diagonal of the covariance matrix) can be added to prevent a singular covariance matrix. This nugget can be specified in advance (if the observations have a fixed uncertainty associated with them), or can be estimated. Estimating the nugget can treat the nugget as a hyperparameter that can be optimised, or find the nugget adaptively by attempting to make the nugget as small as necessary in order to invert the covariance matrix. In practice, the adaptive nugget is the most robust, but requires additional computational effort as the matrix is factored multiple times during each optimisation step.

4.2.1 Covariance Functions

The library implements two stationary *covariance functions*: *Squared Exponential* and *Matern 5/2*. These can be specified when creating a new emulator.

4.2.2 Mean Functions

A *Mean Function* can be specified using R-style *formulas*. By default, these are parsed with the `patsy` library (if it is installed), so R users are encouraged to install this pacakge. However, `mogp_emulator` has its own built-in parser to construct mean functions from a string. Mean functions can also be constructed directly from a rich language mean function classes.

4.2.3 Hyperparameters

There are two types of hyperparameters for a GP emulator, those associated with mean functions, and those associated with the covariance kernel. All mean function hyperparameters are treated on a linear scale, while the kernel hyperparameters are on a logarithmic scale as all kernel parameters are constrained to be positive. The first part of the hyperparameter array contains the n_{mean} mean function parameters (i.e. if the mean function has 4 parameters, then the first 4 entries in the hyperparameter array belong to the mean function), then come the D correlation length hyperparameters (the same as the number of inputs), followed by the covariance and nugget hyperparameters. This means that the total number of hyperparameters depends on the mean function specification and is $n_{mean} + D + 2$.

To interpret the correlation length hyperparameters, the relationship between the reported hyperparameter θ and the correlation length d is $\exp(-\theta) = d^2$. Thus, a large positive value of θ indicates a small correlation length, and a large negative value of θ indicates a large correlation length.

The covariance scale σ^2 can be interpreted as $\exp(\theta) = \sigma^2$. Thus, in this case a large positive value of θ indicates a large overall variance scale and a large negative value of θ indicates a small variance scale.

If the nugget is estimated via hyperparmeter optimisation, the nugget is determined by $\exp(\theta) = \delta$, where δ is added to the diagonal of the covariance matrix. Large positive values of θ indicates a large nugget and a large negative value of θ indicates a small nugget. The nugget value can always be extracted on a linear scale via the `nugget` attribute of a GP regardless of how it was fit, so this is the most reliable way to determine the nugget.

4.2.4 Hyperparameter Priors

Prior beliefs can be specified on hyperparameter values. Exactly how these are interpreted depends on the type of hyperparameter and the type of prior distribution. For *normal prior* distributions, these are applied directly to the hyperparameter values with no transformation. Thus, for mean function hyperparameters, a normal distribution is assumed for a normal prior, while for kernel parameters a lognormal distribution is assumed.

For the *Gamma* and *Inverse Gamma* priors, the distribution is only defined over positive hyperparameter values, so all parameters are exponentiated and then the exponentiated value is used when computing the log PDF.

4.3 Multi-Output GP

Simulations with multiple outputs can be fit by assuming that each output is fit by an independent emulator. The code allows this to be done in parallel using the Python multiprocessing library. This is implemented in the *MultiOutputGP class*, which exhibits an interface that is nearly identical to that of the main *GaussianProcess* class.

4.4 Estimating Hyperparameters

For regular and Multi-Output GPs, hyperparameters are fit using the `fit_GP_MAP` function in the *fitting module*, using L-BFGS optimisation on the negative log posterior. This modifies the hyperparameter values of the GP or MOGP object, returning a fit object that can be used for prediction.

4.5 History Matching

The final component of the UQ workflow is the calibration method. This library implements *History Matching* to perform model calibration to determine which points in the input space are plausible given a set of observations. Performing History Matching requires a fit GP emulator to a set of simulator runs and an observation associated with the simulator output. The emulator is then used to efficiently estimate the simulator output, accounting for all uncertainties, to compare with observations and points that are unlikely to produce the observation can then be “ruled out” and deemed implausible, reducing the input space to better understand the system under question.

At the moment, History Matching is only implemented for a single output and a single set of simulation runs. Future work will extend this to multiple outputs and multiple waves of simulations.

CHAPTER 5

Gaussian Process Demo (Python)

This demo illustrates some various examples of fitting a GP emulator to results of the projectile problem discussed in the *Tutorial*. It shows a few different ways of estimating the hyperparameters. The first two use Maximum Likelihood Estimation with two different kernels (leading to similar performance), while the third uses a linear mean function and places prior distributions on the hyperparameter values. The MAP estimation technique generally leads to significantly better performance for this problem, illustrating the benefit of setting priors.

```
import numpy as np
import mogp_emulator
from projectile import simulator, print_results

# additional GP examples using the projectile demo

# define some common variables

n_samples = 20
n_preds = 10

# Experimental design -- requires a list of parameter bounds if you would like to use
# uniform distributions. If you want to use different distributions, you
# can use any of the standard distributions available in scipy to create
# the appropriate ppf function (the inverse of the cumulative distribution).
# Internally, the code creates the design on the unit hypercube and then uses
# the distribution to map from [0,1] to the real parameter space.

ed = mogp_emulator.LatinHypercubeDesign([(-5., 1.), (0., 1000.)])

# sample space

inputs = ed.sample(n_samples)

# run simulation

targets = np.array([simulator(p) for p in inputs])
```

(continues on next page)

(continued from previous page)

```
###################################################################################

# First example -- fit GP using MLE and Squared Exponential Kernel and predict

print("Example 1: Basic GP")

# create GP and then fit using MLE

gp = mogp_emulator.GaussianProcess(inputs, targets)

gp = mogp_emulator.fit_GP_MAP(gp)

# create 20 target points to predict

predict_points = ed.sample(n_preds)

means, variances, derivs = gp.predict(predict_points)

print_results(predict_points, means)

###################################################################################

# Second Example: How to change the kernel, use a fixed nugget, and create directly␣
↪using fitting function

print("Example 2: Matern Kernel")

# you can simply pass the args to GP to the fitting function

gp_matern = mogp_emulator.fit_GP_MAP(inputs, targets, kernel='Matern52', nugget=1.e-8)

# return type from predict method is an object with mean, unc, etc defined as␣
↪attributes

pred_res = gp_matern.predict(predict_points)

print_results(predict_points, pred_res.mean)

###################################################################################

# Third Example: Specify a mean function and set priors to Fit Hyperparameters via MAP

print("Example 3: Mean Function and MAP fitting")

# This example uses a linear mean function and sets priors on the hyperparameters

# Linear mean has 3 hyperparameters (intercept and 2 slopes, one for each input)
# Kernel has 3 hyperparameters (2 correlation lengths, 1 covariance scale)
# Nugget is the final hyperparameter (7 in total)

# Use a normal prior on all mean function values (requires mean, std)
# Use a normal prior on correlation lengths (which are on a log scale, so becomes a␣
↪lognormal
# distribution once raw values on log scale are converted to linear scale)
# Inverse Gamma distribution on covariance (favors large values)
# Gamma distribution on nugget (favors negative values)
```

(continues on next page)

(continued from previous page)

```
priors = [mogp_emulator.Priors.NormalPrior(0., 10),
          mogp_emulator.Priors.NormalPrior(0., 10.),
          mogp_emulator.Priors.NormalPrior(0., 10.),
          mogp_emulator.Priors.NormalPrior(0., 1.),
          mogp_emulator.Priors.NormalPrior(-10., 1.),
          mogp_emulator.Priors.InvGammaPrior(1., 1.),
          mogp_emulator.Priors.GammaPrior(1., 1.)]

# create GP, passing list of priors and a string representing the mean function
# tell it to estimate the nugget as well

gp_map = mogp_emulator.GaussianProcess(inputs, targets, mean="x[0]+x[1]",␣
↪priors=priors, nugget="fit")

# fit hyperparameters

gp_map = mogp_emulator.fit_GP_MAP(gp_map)

# gp can be called directly if only the means are desired

pred_means = gp_map(predict_points)

print_results(predict_points, pred_means)
```

CHAPTER 6

Mutual Information for Computer Experiments (MICE) Demos

This demo shows how to use the MICEDesign class to run a sequential experimental design. The predictions of a GP on some test points is compared between an LHD and the MICE design, showing that the performance of the MICE design is a significant improvement.

```
import mogp_emulator
import numpy as np
from projectile import simulator, print_results

# simple MICE examples using the projectile demo

# Base design -- requires a list of parameter bounds if you would like to use
# uniform distributions. If you want to use different distributions, you
# can use any of the standard distributions available in scipy to create
# the appropriate ppf function (the inverse of the cumulative distribution).
# Internally, the code creates the design on the unit hypercube and then uses
# the distribution to map from [0,1] to the real parameter space.

lhd = mogp_emulator.LatinHypercubeDesign([(-5., 1.), (0., 1000.)])

###############################################################################

# first example -- run entire design internally within the MICE class.

# first argument is base design (required), second is simulator function (optional,
# but required if you want the code to run the simualtions internally)

# Other optional arguments include:
# n_samples (number of sequential design steps, optional, default is not specified
# meaning that you will specify when running the sequential design)
# n_init (size of initial design, default 10)
# n_cand (number of candidate points, default is 50)
# nugget (nugget parameter for design GP, default is to set adaptively)
# nugget_s (nugget parameter for candidate GP, default is 1.)
```

(continues on next page)

(continued from previous page)

```
n_init = 5
n_samples = 20
n_cand = 100

md = mogp_emulator.MICEDesign(lhd, simulator, n_samples=n_samples, n_init=n_init, n_
↪cand=n_cand)

md.run_sequential_design()

# get design and outputs

inputs = md.get_inputs()
targets = md.get_targets()

print("Example 1:")
print("Design inputs:\n", inputs)
print("Design targets:\n", targets)
print()

##################################################################################

# second example: run design manually

md2 = mogp_emulator.MICEDesign(lhd, n_init=n_init, n_cand=n_cand)

init_design = md2.generate_initial_design()

print("Example 2:")
print("Initial design:\n", init_design)

# run initial points manually

init_targets = np.array([simulator(s) for s in init_design])

# set initial targets

md2.set_initial_targets(init_targets)

# run 20 sequential design steps

for d in range(n_samples):
    next_point = md2.get_next_point()
    next_target = simulator(next_point)
    md2.set_next_target(next_target)

# look at design and outputs

inputs = md2.get_inputs()
targets = md2.get_targets()

print("Final inputs:\n", inputs)
print("Final targets:\n", targets)

# look at final GP emulator and make some predictions to compare with lhd

lhd_design = lhd.sample(n_init + n_samples)
```

(continues on next page)

(continued from previous page)

```
gp_lhd = mogp_emulator.fit_GP_MAP(lhd_design, np.array([simulator(p) for p in lhd_
↪design]))

gp_mice = mogp_emulator.GaussianProcess(inputs, targets)

gp_mice = mogp_emulator.fit_GP_MAP(inputs, targets)

test_points = lhd.sample(10)

print("LHD:")
print_results(test_points, gp_lhd(test_points))
print()
print("MICE:")
print_results(test_points, gp_mice(test_points))
```

CHAPTER 7

Gaussian Process Demo (R)

```
# Short demo of how to fit and use the GP class to predict unseen values based on a
# mean function and prior distributions.

# Before loading reticulate, you will need to configure your Python Path to
# use the correct Python version where mogp_emulator is installed.
# mogp_emulator requires Python 3, but some OSs still have Python 2 as the
# default, so you may not get the right one unless you explicitly configure
# it in reticulate. I use the Python that I installed on my Mac with homebrew,
# though on Linux the Python installed via a package manager may have a
# different path.

# The environment variable is RETICULATE_PYTHON, and I set it to
# "/usr/local/bin/python" as this is the Python where mogp_emulator is installed.
# This is set automatically in my .Renviron startup file in my home directory,
# but you may want to configure it some other way. No matter how you decide
# to configure it, you have to set it prior to loading the reticulate library.

library(reticulate)

mogp_emulator <- import("mogp_emulator")
mogp_priors <- import("mogp_emulator.Priors")

# create some data

n_train <- 10
x_scale <- 2.
x1 <- runif(n_train)*x_scale
x2 <- runif(n_train)*x_scale
y <- exp(-x1**2 - x2**2)
x <- data.frame(x1, x2, y)

# GaussianProcess requires data as a matrix, but often you may want to do some
# regression using a data frame in R. To do this, we can split this data frame
# into inputs, targets, and a dictionary mapping column names to integer indices
```

(continues on next page)

(continued from previous page)

```
# using the function below

extract_targets <- function(df, target_cols = list("y")) {
  "separate a data frame into inputs, targets, and inputdict for use with GP class"

  for (t in target_cols) {
    stopifnot(t %in% names(x))
  }

  n_targets <- length(target_cols)

  inputs <- matrix(NA, ncol=ncol(x) - n_targets, nrow=nrow(x))
  targets <- matrix(NA, ncol=n_targets, nrow=nrow(x))
  inputdict <- dict()

  input_count <- 1
  target_count <- 1

  for (n in names(x)) {
    if (n %in% target_cols) {
      targets[,target_count] <- as.matrix(x[n])
    } else {
      inputs[,input_count] <- as.matrix(x[n])
      inputdict[n] <- as.integer(input_count - 1)
      input_count <- input_count + 1
    }
  }

  if (n_targets == 1) {
    targets <- c(targets)
  }

  return(list(inputs, targets, inputdict))
}

target_list <- extract_targets(x)
inputs <- target_list[[1]]
targets <- target_list[[2]]
inputdict <- target_list[[3]]

# Create the mean function formula as a string (or you could extract from the
# formula found via regression). If you want correct expansion of your formula
# in the Python code, you will need to install the patsy package (it is pip
# installable) as it is used internally in mogp_emulator to parse formulas.

# Additionally, you will need to convert the column names from the data frame
# to integer indices in the inputs matrix. This is done with a dict object as
# illustrated below.

mean_func <- "y ~ x1 + x2 + I(x1*x2)"

# Priors are specified by giving a list of prior objects (or NULL if you
# wish to use weak prior information). Each distribution has some parameters
# to set -- NormalPrior is (mean, std), Gamma is (shape, scale), and
# InvGammaPrior is (shape, scale). See the documentation or code for the exact
# functional format of the PDF.
```

(continues on next page)

(continued from previous page)

```
# If you don't know how many parameters you need to specify, it depends on
# the mean function and the number of input dimensions. Mean functions
# have a fixed number of parameters (though in some cases this can depend
# on the dimension of the inputs as well), and then covariance functions have
# one correlation length per input dimension plus a covariance scale and
# a nugget parameter.

# If in doubt, you can create the GP instance with no priors, use gp$n_params
# to get the number, and then set the priors manually using gp$priors <- priors

# In this case, we have 4 mean function parameters (normal distribution on a
# linear scale), 2 correlations lengths (normal distribution on a log scale,
# so lognormal), a sigma^2 covariance parameter (inverse gamma) and a nugget
# (Gamma). If you choose an adaptive or fixed nugget, the nugget prior is ignored.

priors <- list(mogp_priors$NormalPrior(0., 1.),
               mogp_priors$NormalPrior(0., 1.),
               mogp_priors$NormalPrior(0., 1.),
               mogp_priors$NormalPrior(0., 1.),
               mogp_priors$NormalPrior(0., 1.),
               mogp_priors$NormalPrior(0., 1.),
               mogp_priors$InvGammaPrior(2., 1.),
               mogp_priors$GammaPrior(1., 0.2))

# Finally, create the GP instance. If we had multiple outputs, we would
# create a MultiOutputGP class in a similar way, but would have the option
# of giving a single mean and list of priors (assumes it is the same for
# each emulator), or a list of mean functions and a list of lists of
# prior distributions. nugget can also be set with a single value or a list.

gp <- mogp_emulator$GaussianProcess(inputs, targets,
                                    mean=mean_func,
                                    priors=priors,
                                    nugget="fit",
                                    inputdict=inputdict)

# gp is fit using the fit_GP_MAP function. It accepts a GaussianProcess or
# MultiOutputGP object and returns the same type of object with the
# hyperparameters fit via MAP estimation, with some options for how to perform
# the minimization routine. You can also pass the arguments to create a GP/MOGP
# to this function and it will return the object with estimated hyperparameters

gp <- mogp_emulator$fit_GP_MAP(gp)

print(gp$current_logpost)
print(gp$theta)

# now create some test data to make predictions and compare with known values

n_test <- 10000

x1_test <- runif(n_test)*x_scale
x2_test <- runif(n_test)*x_scale

x_test <- cbind(x1_test, x2_test)
y_actual <- exp(-x1_test**2 - x2_test**2)
```

(continues on next page)

(continued from previous page)

```
y_predict <- gp$predict(x_test)

# y_predict is an object holding the mean, variance and derivatives (if computed)
# access the values via y_predict$mean, y_predict$unc, and y_predict$deriv

print(sum((y_actual - y_predict$mean)**2)/n_test)
```

CHAPTER 8

Uncertainty Quantification Methods

8.1 The MUCM Toolkit, release 6

Welcome to the *MUCM* Toolkit. The toolkit is a resource for people interested in quantifying and managing uncertainty in the outputs of mathematical models of complex real-world processes. We refer to such a model as a simulation model or a *simulator*.

The toolkit is a large, interconnected set of webpages and one way to use it is just to browse more or less randomly through it. However, we have also provided some organised starting points and threads through the toolkit.

- We have an introductory tutorial on MUCM methods and uncertainty in simulator outputs *here*.
- You can read about the *toolkit structure*.
- The various threads, each of which sets out in a series of steps how to use the MUCM approach to build an *emulator* of a simulator and to use it to address some standard problems faced by modellers and users of simulation models. This release contains the following threads:
  - *ThreadCoreGP*, which deals with the simplest emulation scenario, called the *core problem*, using the *Gaussian process* approach;
  - *ThreadCoreBL*, which also deals with the core problem, but follows the *Bayes linear* approach. A simple guide to the differences between the two approaches can be found in the alternatives page on Gaussian process or Bayes Linear Emulator (*AltGPorBLEmulator*);
  - *ThreadVariantMultipleOutputs*, which extends the core problem to address the case where we wish to emulate two or more simulator outputs;
  - *ThreadVariantDynamic*, which extends the core analysis in a different direction, where we wish to emulate the time series output of a dynamic simulator;
  - *ThreadVariantTwoLevelEmulation*, which considers the situation where we have two simulators of the same real-world phenomenon, a slow but relatively accurate simulator whose output we wish to emulate, and a quick and relatively crude simulator. This thread discusses how to use many runs of the fast simulator to build an informative prior model for the slow simulator, so that fewer training runs of the slow simulator are needed;

- *ThreadVariantWithDerivatives*, which extends the core analysis for the case where we can obtain derivatives of the simulator output with respect to the various inputs, to use as training data;
- *ThreadVariantModelDiscrepancy*, which deals with modelling the relationship between the simulator outputs and the real-world process being simulated. Recognising this *model discrepancy* is a crucial step in making useful predictions from simulators, in calibrating simulation models and handling multiple models.
- *ThreadGenericMultipleEmulators*, which deals with combining two or more emulators to produce emulation of some combination of the respective simulator outputs;
- *ThreadGenericEmulateDerivatives*, which shows how to use an emulator to predict the values of derivatives of the simulator output;
- *ThreadGenericHistoryMatching*, which deals with iteratively narrowing down the region of possible input values for which the simulator would produce outputs acceptably close to observed data. This topic is related to calibration, which will be addressed in a future release of the toolkit.
- *ThreadTopicSensitivityAnalysis*, which is a topic thread providing more detailed background on the topic of sensitivity analysis, and linking together the various procedures for such techniques in the other toolkit threads.
- *ThreadTopicScreening*, which provides a broad view of the idea of *screening* the simulator inputs to reduce their dimensionality.
- *ThreadTopicExperimentalDesign*, which gives a detailed overview of the methods of experimental design that are relevant to MUCM, particularly those relating to the design of a training sample.

Later releases of the toolkit will add more threads and other material, including more extensive examples to guide the toolkit user and further Case Studies. In each release we also add more detail to some of the existing threads; for instance, in this release we have a substantial reworking of the variant thread on emulating multiple outputs, and also new material on variance learning for Bayes Linear emulation.

8.2 Meta-pages: Toolkit tutorial

8.2.1 Uncertainty in models

Modelling is a vital part of research and development in almost every sphere of modern life. The objective of *MUCM* is to develop a technology that is capable of addressing all sources of uncertainty in model predictions and to quantify their implications efficiently, even in the most complex models. It has the potential to revolutionise scientific debate by resolving the contradictions in competing models. It will also have a radical effect on everyday modelling and model usage by making the uncertainties in model outputs transparent to modellers and end users alike.

Those who rely on models to understand complex processes, and for prediction, optimisation and many kinds of decision- and policy-making, increasingly wish to know how much they can trust the model outputs. Uncertainty and inaccuracy in the outputs arises from numerous sources, including error in initial conditions, error in model parameters, imperfect science in the model equations, approximate solutions to model equations and errors in model structure or logic. The nature and magnitudes of these contributory uncertainties are often very difficult to estimate, but it is vital to do so. All the while, for instance, different models produce very different predictions of the magnitude of global warming effects, with no credible error bounds, sceptics can continue to ignore them and pressure groups will seize upon the most pessimistic predictions.

Even if we can quantify all uncertainties in model inputs and structure, it is a complex task to derive appropriate measures of output uncertainty. One well-established methodology to address this problem of *uncertainty analysis*, or probabilistic sensitivity analysis, is to propagate input uncertainty through the model by Monte Carlo. However, this requires making typically tens of thousands of runs of the model, each with different randomly sampled inputs, and this is impractical for complex models. For any model that takes more than a few seconds of computer time per run, a thorough Monte Carlo *sensitivity analysis* becomes infeasible.

MUCM focuses on new methods which are orders of magnitude more efficient than Monte Carlo, requiring typically just a few hundreds of model runs, thereby providing very significant productivity gains for the researchers or analysis teams involved. Furthermore, these methods can address several related, but more demanding tasks that are of importance to modellers and model users, usually without requiring more model runs.

- Calibration and data assimilation. *Calibration*, the process of adjusting uncertain model parameters to fit the model to observed data, is typically a very demanding task that can involve many man months or even years of effort. Data assimilation, in which data are used to adjust the state vector of a dynamic model, is equally demanding and the subject of quite intensive research in its own right. MUCM methods can not only perform these tasks more efficiently, but also properly characterise how they reduce uncertainty about those parameters and state vector (and hence reduce uncertainty in model outputs).
- Variance-based sensitivity analysis and value of information. These are tools to explore how the model responds to changes in inputs, and the contribution that each uncertain input makes to the overall output uncertainty. They give modellers insight into how their models behave (often pointing to bugs in coding or model failures) and allow model users to prioritise research to reduce uncertainty.
- Validation and structural uncertainty. It is often said that models cannot be *validated* since no model is perfect. Nevertheless, it is possible to validate the combination of a model with a description of uncertainty, simply by computing implied probability distributions for test data and then verifying that they lie within the bounds of those distributions. However, this requires all forms of uncertainty to be accounted for, including uncertainty in model structure, and cannot be addressed by conventional Monte Carlo analyses. MUCM methods are able to tackle this problem, and indeed a model for model discrepancy underlies the calibration techniques.

8.2.2 Emulation

We refer to the process model to which MUCM methods are applied (and its implementation in computer code) as the *simulator*. The key to MUCM technology is the idea of an *emulator*, which is a statistical representation of knowledge and beliefs about the simulator. The emulator is not just a very accurate approximation to the simulator itself, but also incorporates a statistically validated description of its own accuracy. It is created using a set of runs of the simulator, in which the simulator outputs are observed at different points in the input space. This is called the *training sample*.

Emulator for a simple model based on 3 data points

Figure 1 illustrates how the emulator works in an extremely simple case. The simulator is supposed to be a function of a single input x. Three training runs have been performed and the output computed at $x = 1, 3$ and 5; these are the three points marked in the figure. The solid line is the emulator mean, which is a prediction of what the simulator would produce if run at any other value of x. The dotted lines give 95% probability intervals for those predictions. Notice that the emulator interpolates the training data precisely and correctly shows no uncertainty at those points, but the uncertainty increases between training data points.

Figure 2 shows the same example after we add two more training runs. The emulator mean is adjusted to pass through the two new points, and the uncertainty is reduced considerably. Indeed, within the range of the training data the emulator now predicts the simulator output with very small uncertainty.

Properties of the emulator

The same features apply in any application with any number of inputs: the emulator reproduces the training data exactly, interpolating them smoothly with uncertainty that increases between data points. Increasing the number of training data points, so that they are closer together, in principle allows the simulator to be emulated to any desired degree of accuracy, with small uncertainty throughout the region of the input space of interest.

The emulator runs essentially instantaneously, making intensive exploration of the model and the consequences of uncertainty in inputs and model structure feasible for even highly complex models. Its mathematical form is also simple,

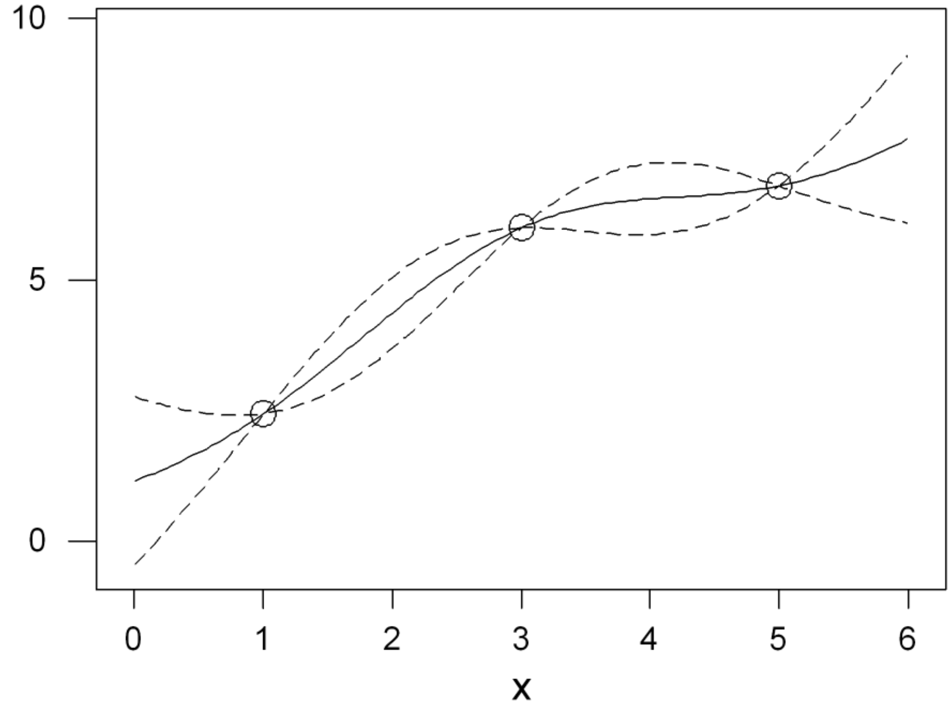


Fig. 1: **Figure 1:** An example of an emulator fit so a small number of training points.

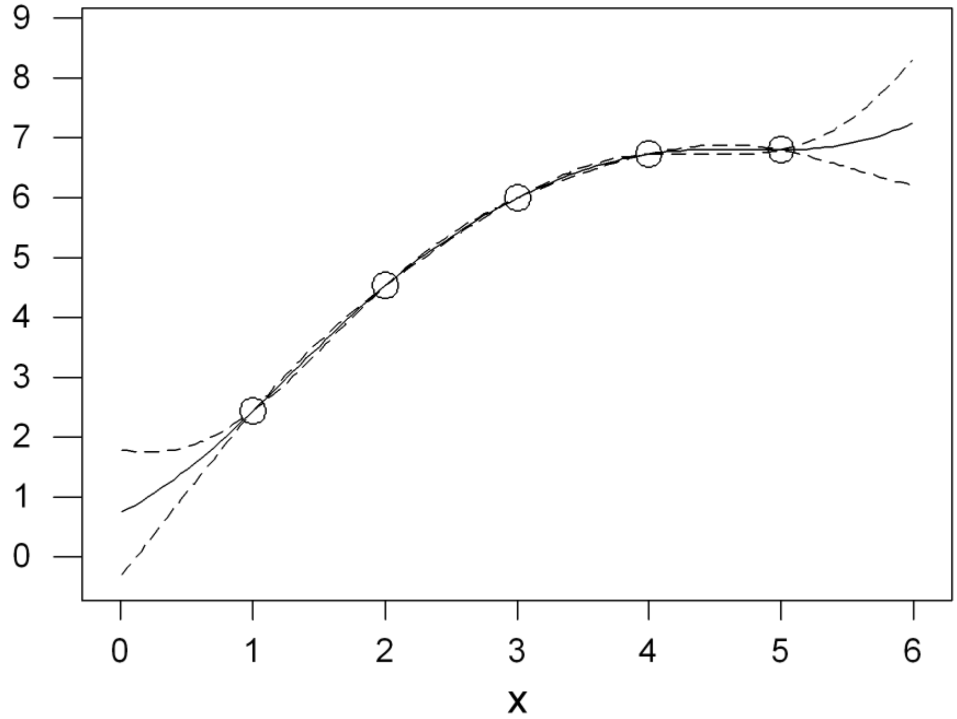


Fig. 2: **Figure 2:** Additional points reduce the uncertainty of the emulator.

so that in many cases the results of complex analyses of simulator output, such as sensitivity analysis, can be predicted analytically without needing to 'run' the emulator. In other situations, the analyses can be performed very much more quickly by running the emulator as a surrogate for the simulator, which may make feasible analyses that would otherwise be impossible because of computational intensity. Traditional approaches to tasks such as uncertainty analysis, particularly those based on Monte Carlo sampling in the input space, have been found in a range of applications to be orders of magnitude less efficient. That is, to achieve comparable accuracy those methods require tens, hundreds or even thousands of times as many simulator runs as MUCM methods based on emulation.

8.2.3 Toolkit issues

Although conceptually simple, the MUCM methods can be complex to apply. The role of the toolkit is to provide straightforward specifications of the MUCM technology. It is aimed at users of process simulators who wish to understand and manage the uncertainties in simulator predictions, and for other researchers in this field. Toolkit "threads" take the user step by step through building emulators and using them to tackle questions such as sensitivity analysis or calibration. In any application, a number of decisions need to be taken concerning questions such as the detailed form of the emulator; here are some of the principal issues that are addressed in toolkit pages.

- Type of emulator. In MUCM two different approaches to emulation are the fully *Bayesian* approach based on *Gaussian process* emulators and the *Bayes linear* approach which does not make distributional assumptions. The alternative approaches have advantages in different situations, but the principal distinction between them is a fundamental difference in how uncertainty is represented.
- *Training sample* design. The creation of a suitable training sample is a key step. This problem raises interesting new challenges for statistical experimental design theory. Although in principle we can emulate the simulator very accurately with a large enough training sample, in practice a large number of simulator runs is often impractical, and careful design is necessary to get the most out of a limited number of training runs.
- Fitting the emulator. Within the theory of the emulator are a number of optional features and parameters to be estimated. There are statistical and computational challenges here that are addressed in the relevant toolkit pages.
- Validating the emulator. The emulator may claim high accuracy in representing the underlying simulator, but is that claim justified? Validation uses a second sample of simulator runs to check the validity of the emulator's claims.
- Model discrepancy. No model is perfect, and no simulator is a perfect representation of the real-world process it simulates. Techniques such as calibration rely on contrasting simulator outputs with real-world observations. It is vital to recognise that the real-world observations differ from the simulator output because of simulator error as well as observation error. An integral part of the MUCM technology is understanding model discrepancy.

8.3 Meta-pages: Notation

This page sets out the notational conventions used in the *MUCM* toolkit.

8.3.1 General Conventions

Vectors and matrices

Vectors in the toolkit can be either row $(1 \times n)$ or column $(n \times 1)$ vectors. When not specified explicitly, a vector is a column vector by default.

We do not identify vectors or matrices by bold face.

Functions

Functions are denoted with a dot argument, e.g. $f(\cdot)$, while their value at a point x is $f(x)$.

The following compact notation is used for arrays of function values. Suppose that $g(\cdot)$ is a function taking values x in some space, and let D be a vector with n elements $D = \{x_1, x_2, \ldots, x_n\}$ in that space; then $g(D)$ is made up of the function values $g(x_1), g(x_2), \ldots, g(x_n)$. More specifically, we have the following cases:

- If $g(x)$ is a scalar, then $g(D)$ is a $(n \times 1)$ (column) vector by default, unless explicitly defined as a $(1 \times n)$ (row) vector.
- If $g(x)$ is a $(t \times 1)$ column vector, then $g(D)$ is a matrix of dimension $t \times n$.
- If $g(x)$ is a $(1 \times t)$ row vector, then $g(D)$ is a matrix of dimension $(n \times t)$.
- If $g(x, x')$ is a scalar, then $g(D, x')$ is a $(n \times 1)$ column vector, $g(x, D')$ is a $(1 \times n)$ row vector and $g(D, D')$ is a $(n \times n)$ matrix.

Other notation

The * superscript denotes a posterior value or function.

The matrix/vector transpose is denoted with a roman superscript $^{\mathrm{T}}$.

Expectations, Variances and Covariances are denoted as $\mathrm{E}[\cdot], \mathrm{Var}[\cdot], \mathrm{Cov}[\cdot, \cdot]$.

The trace of a matrix is denoted as tr.

8.3.2 Reserved symbols

The following is a list of symbols that represent fundamental quantities across the toolkit. These *reserved* symbols will always represent quantities of the same generic type in the toolkit. For instance, the symbol n will always denote a number of *design* points. Where two or more different designs are considered in a toolkit page, their sizes will be distinguished by subscripts or superscripts, e.g. n_t might be the size of a *training sample* design, while n_v is the size of a *validation* design. Notation should always be defined properly in toolkit pages, but the use of reserved symbols has mnemonic value to assist the reader in remembering the meanings of different symbols.

The reserved symbols comprise a relatively small set of symbols (and note that if only a lower-case symbol is reserved the corresponding upper-case symbol is not). Non-reserved symbols have no special meanings in the toolkit.

Symbol	Meaning
Dimensions	
n	Number of *design* points
p	Number of *active inputs*
q	Number of *basis functions*
r	Number of outputs
s	Number of *hyperparameter* sets in an *emulator*
Input - Output	
x	Point in the *simulator*'s input space
y	Reality - the actual system value
z	Observation of reality y
D	Design, comprising an ordered set of points in an input space
$d(\cdot)$	Model discrepancy function
$f(\cdot)$	The output(s) of a simulator
$h(\cdot)$	Vector of basis functions
Hyperparameters	
β	*Hyperparameters* of a *mean* function
δ	Hyperparameters of a *correlation* function
σ^2	Scale hyperparameter for a *covariance* function
θ	Collection of hyperparameters on which the emulator is conditioned
ν	*Nugget*
π	*Distribution of hyperparameters*
Statistics	
$m(\cdot)$	Mean function
$v(\cdot,\cdot)$	Covariance function
$m^*(\cdot)$	Emulator's posterior mean, conditioned on the hyperparameters and design points
$v^*(\cdot,\cdot)$	Emulator's posterior covariance, conditioned on the hyperparameters and design points
$c(\cdot,\cdot)$	Correlation function

8.4 Meta-pages: Comments

The MUCM team is interested to receive your comments on the toolkit. You can send comments about particular pages (to tell us of errors or alternative approaches, or to give us the benefit of your own knowledge and experience with MUCM-related ideas and methods). You can also send us comments about the toolkit in general: (Do you like it? How could we improve it generally? What features would you like to see? How could we organise it better?).

Please send your comments to the webmaster.

8.5 Meta-pages: Toolkit Structure

8.5.1 Main threads

The toolkit contains a large number of pages, and there are several ways to use it. One would be just to browse - pick a page from this *page list* and just follow connections - but we have also created more structured entry points. The first of these is the set of main threads. These will take the user through the complete process of a *MUCM* application - designing and building an *emulator* to represent a *simulator*, and then using it to carry out tasks such as *sensitivity analysis* or *calibration*.

The simplest main threads are the core threads. There are two of these, one for the *Gaussian Process* (GP, or fully

Bayesian) version of the core, and one for the *Bayes Linear* (BL) version. The underlying differences between these two approaches are discussed in the alternatives page on Gaussian Process or Bayes Linear based emulator (*AltGPorBLEmulator*). In both cases, the core thread deals with developing emulators for a single output of a simulator that is *deterministic*. Further details of the core model are given in the threads themselves, *ThreadCoreGP* and *ThreadCoreBL*.

Further main threads deal with variations on the basic core model. In principle, these threads will try to handle both GP and BL approaches. However, there will be various components of these threads where the relevant tools have only been developed fully for one of the two approaches, and methods for the other approach will only be discussed in general terms.

Other main threads will address generic extensions that apply to both core and variant threads.

The main threads are currently planned to be as follows:

- *ThreadCoreGP* - the core model, dealt with by fully Bayesian, Gaussian Process, emulation
- *ThreadCoreBL* - the core model, dealt with by Bayes Linear emulation
- *ThreadVariantMultipleOutputs* - a variant of the core model in which we emulate more than one output of a simulator
- *ThreadVariantDynamic* - a special case of multiple outputs is when the simulator outputs are generated iteratively as a time series
- *ThreadVariantTwoLevelEmulation* - a variant in which a fast simulator is used to help build an emulator of a slow simulator
- ThreadVariantMultipleSimulators - variant of the core model in which we emulate outputs from more than one related simulator, a special case of which is when the real world is regarded as a perfect simulator
- ThreadVariantStochastic - variant of the core model in which the simulator output is *stochastic*
- *ThreadVariantWithDerivatives* - variant of the core model in which we include derivative information
- *ThreadVariantModelDiscrepancy* - a variant that deals with modelling the relationship between the simulator outputs and the real-world process being simulated
- *ThreadGenericMultipleEmulators* - a thread showing how to combine independent emulators (for outputs of different simulators or different outputs of one simulator) to address tasks relating to combinations of the outputs being simulated
- *ThreadGenericEmulateDerivatives* - a thread showing how to emulate derivatives of outputs
- *ThreadGenericHistoryMatching* - a thread using observations of the real system to learn about the inputs of the model

It should be noted that simulators certainly exist in which we need to take account of more than one of the variations. For instance, we may be interested in multiple outputs which are also stochastic. Where the relevant tools have been developed, they will be included through cross-linkages between the main threads, or by additional main threads.

The toolkit is being released to the public in stages, so that only some of the threads are currently available. Others will be included in future releases.

8.5.2 Topic Threads

Another way to use the toolkit is provided through a number of topic threads. Whereas the various main threads take the user through the process of modelling, building an emulator and using that emulator to carry out relevant tasks, a topic thread focusses on a particular aspect of that process. For instance, a topic thread could deal with issues of modelling, and would describe the core model, variants on it associated with the other main threads, and go on to more complex variants for which tools are as yet unavailable or are under development. Another topic thread might consider the design of training samples for building an emulator, or the task of sensitivity analysis.

Topic threads provide a technical background to their topics that is common across different kinds of emulators. They allow toolkit users to gain a more in-depth understanding and to appreciate relationships between how the topic is addressed in different main threads. Whereas the main threads are aimed at toolkit users who wish to apply the MUCM tools, the topic threads will be intended more for researchers in the field or for users who want to gain a deeper understanding of the tools.

Topic threads now available or under development include:

- *ThreadTopicSensitivityAnalysis*
- *ThreadTopicScreening*
- *ThreadTopicExperimentalDesign*

8.5.3 Other Page Types

Apart from the Threads, the other pages of the Toolkit belong to one of the following categories

- Procedure - The description of an operation or an algorithm. Procedure pages should provide sufficient information to allow the implementation of the operation that is being described.
- Discussion - Pages that discuss issues that may arise during the implementation of a method, or other optional details.
- Alternatives - These pages present available options when building a specific part of an emulator (e.g. choosing a covariance function) and provide some guidance for doing the selection.
- Definition - Definition of a term or a concept.
- Example - A page that provides a worked example of a thread or procedure.
- Meta - Any page that does not fall in one of the above categories, usually pages about the Toolkit itself.

Page types are identifiable by the start of the page name - Thread, Proc, Disc, Alt, Def, Exam or Meta.

8.6 Meta-pages: Page List

This is a complete list of the pages that will comprise the sixth release. Pages that have been modified since release 5 are marked * , while pages that are new in release 6 are marked **.

8.6.1 Generic pages

- *AltGPorBLEmulator*
- *MetaComments*
- *MetaCopyrightNotice*
- *MetaHomePage* *
- *MetaNotation*
- *MetaSoftwareDisclaimer*
- *MetaToolkitPageList* *
- *MetaToolkitStructure* *
- *MetaToolkitTutorial*

8.6.2 Definition/Glossary pages

- *DefActiveInput*
- *DefAdjoint*
- *DefAssessment*
- *DefBasisFunctions*
- *DefBayesian*
- *DefBayesLinear*
- *DefBestInput*
- *DefBLAdjust*
- *DefBLVarianceLearning* *
- *DefCalibration*
- *DefCodeUncertainty*
- *DefConjugate*
- *DefCorrelationLength*
- *DefDataAssimilation*
- *DefDecisionBasedSA*
- *DefDesign*
- *DefDeterministic*
- *DefDynamic*
- *DefElicitation*
- *DefEmulator*
- *DefExchangeability* **
- *DefForcingInput*
- *DefGP*
- *DefHistoryMatching* **
- *DefHyperparameter*
- *DefImplausibilityMeasure* **
- *DefInactiveInput*
- *DefModelBasedDesign***
- *DefModelDiscrepancy*
- *DefMUCM*
- *DefMultilevelEmulation*
- *DefMultivariateGP* *
- *DefMultivariateTProcess* *
- *DefNugget*
- *DefPrincipalComponentAnalysis*

- *DefProper*
- *DefRegularity*
- *DefReification*
- *DefScreening*
- *DefSecondOrderExch* **
- *DefSecondOrderSpec*
- *DefSensitivityAnalysis*
- *DefSeparable*
- *DefSimulator*
- *DefSingleStepFunction*
- *DefSmoothingKernel* **
- *DefSmoothness*
- *DefSpaceFillingDesign***
- *DefStateVector*
- *DefStochastic*
- *DefTProcess*
- *DefTrainingSample*
- *DefUncertaintyAnalysis*
- *DefValidation*
- *DefVarianceBasedSA*
- *DefWeakPrior*

8.6.3 GP core thread

- *ThreadCoreGP* *
- *AltCoreDesign* *
- *AltCorrelationFunction*
- *AltEstimateDelta*
- *AltGPPriors*
- *AltMeanFunction* *
- *DiscActiveInputs*
- *DiscBuildCoreGP*
- *DiscCore*
- *DiscCoreDesign* *
- *DiscCoreValidationDesign*
- *DiscCovarianceFunction*
- *DiscGaussianAssumption*

- *DiscGPBasedEmulator*
- *DiscPostModeDelta*
- *DiscRealisationDesign*
- *DiscUANugget*
- *ExamCoreGP1Dim*
- *ExamCoreGP2Dim*
- *ProcApproxDeltaPosterior*
- *ProcBuildCoreGP*
- *ProcHaltonDesign*
- *ProcLatticeDesign*
- *ProcLHC*
- *ProcMCMCDeltaCoreGP*
- *ProcOptimalLHC*
- *ProcOutputTransformation*
- *ProcPivotedCholesky*
- *ProcPredictGP*
- *ProcSimulationBasedInference*
- *ProcUAGP*
- *ProcValidateCoreGP*
- *ProcVarSAGP*
- *ProcWeylDesign*

8.6.4 BL core thread

- *ThreadCoreBL* *
- *AltBasisFunctions*
- *AltBLPriors*
- *DiscAdjustExchBeliefs* **
- *DiscBayesLinearTheory*
- *DiscStructuredMeanFunction*
- *ProcBLAdjust*
- *ProcBLPredict*
- *ProcBuildCoreBL*
- *ProcUABL*
- *ProcVariogram*
- *ProcBLVarianceLearning* **

8.6.5 Multiple outputs variant thread

- *ThreadVariantMultipleOutputs* *
- *AltMeanFunctionMultivariate* **
- *AltMultipleOutputsApproach* *
- *AltMultivariateCovarianceStructures* **
- *AltMultivariateGPPriors* *
- *ExamMultipleOutputs* **
- *ExamMultipleOutputsPCA* *
- *ProcBuildMultiOutputGP* *
- *ProcBuildMultiOutputGPSep* **
- *ProcOutputsPrincipalComponents*
- *ProcPredictMultiOutputFunction* **

8.6.6 Dynamic emulation variant thread

- *ThreadVariantDynamic*
- *AltDynamicEmulationApproach*
- *AltIteratingSingleStepEmulators*
- *DiscUncertaintyAnalysis*
- *DiscMonteCarlo*
- *ProcApproximateIterateSingleStepEmulator*
- *ProcApproximateUpdateDynamicMeanandVariance*
- *ProcExactIterateSingleStepEmulator*
- *ProcExploreFullSimulatorDesignRegion*
- *ProcOutputSample*
- *ProcUADynamicEmulator*
- *ProcUpdateDynamicMeanAndVariance*

8.6.7 Two-level emulation variant thread

- *ThreadVariantTwoLevelEmulation*
- *ProcBuildCoreBLEmpirical*

8.6.8 Derivatives variant and generic threads

- *ThreadVariantWithDerivatives*
- *ProcBuildWithDerivsGP*
- *ExamVariantWithDerivatives1Dim*

- *ThreadGenericEmulateDerivatives*
- *ProcBuildEmulateDerivsGP*

8.6.9 Linking Models to Reality Variant Thread

- *ThreadVariantModelDiscrepancy*
- *DiscWhyModelDiscrepancy*
- *DiscBestInput*
- *DiscObservations*
- *DiscExpertAssessMD*
- *DiscInformalAssessMD*
- *DiscFormalAssessMD*
- *DiscStructuredMD*
- *DiscReification*
- *DiscReificationTheory*
- *DiscExchangeableModels*

8.6.10 Multiple emulators generic thread

- *ThreadGenericMultipleEmulators*
- *ProcPredictMultipleEmulators*
- *ProcUAMultipleEmulators*

8.6.11 History Matching Generic Thread

- *ThreadGenericHistoryMatching***
- *AltImplausibilityMeasure***
- *DiscImplausibilityCutoff***
- *DiscIterativeRefocussing***
- *Exam1DHistoryMatch***

8.6.12 Topic thread on sensitivity analysis

- *ThreadTopicSensitivityAnalysis*
- *DiscDecisionBasedSA*
- *DiscSensitivityAndDecision*
- *DiscSensitivityAndOutputUncertainty*
- *DiscSensitivityAndSimplification*
- *DiscToolkitSensitivityAnalysis*

- *DiscVarianceBasedSA*
- *DiscVarianceBasedSATheory*
- *DiscWhyProbabilisticSA*
- *ExamDecisionBasedSA*

8.6.13 Topic thread on screening

- *ThreadTopicScreening*
- *AltScreeningChoice*
- *ExamScreeningAutomaticRelevanceDetermination*
- *ExamScreeningMorris*
- *ProcAutomaticRelevanceDetermination*
- *ProcDataPreProcessing*
- *ProcMorris*

8.6.14 Topic thread on Design of Experiments

- *ThreadTopicExperimentalDesign***
- *AltNumericalSolutionForKarhunenLoeveExpansion***
- *AltOptimalCriteria***
- *AltOptimalDesignAlgorithms***
- *DiscFactorialDesign***
- *DiscKarhunenLoeveExpansion***
- *DiscSobol* **
- *ProcASCM***
- *ProcBranchAndBoundAlgorithm***
- *ProcExchangeAlgorithm***
- *ProcFourierExpansionForKL***
- *ProcHaarWaveletExpansionForKL***
- *ProcSobolSequence***

8.7 Meta-pages: Copyright Notice

The MUCM toolkit is produced by the MUCM project and copyright in all the toolkit pages is held on behalf of the participating institutions (University of Sheffield, Durham University, Aston University, London School of Economics and the National Oceanography Centre, Southampton) by the MUCM Management Board.

Pages may not be copied in whole or in part or quoted from without full acknowledgement being made to the MUCM toolkit. The URL(s) of the page(s) concerned must be included in all such acknowledgements.

Further information, requests and enquiries concerning reproduction and rights should be addressed to:

Managing Uncertainty in Complex Models
Department of Probability & Statistics
University of Sheffield
Hicks Building
Hounsfield Road
Sheffield
S3 7RH
UK
Tel : +44 (0) 114 222 3753
E mail: mucm@sheffieldNOSPAM.ac.uk

Disclaimer

Information at this site is general information provided as part of the RCUK Managing Uncertainty in Complex Models project. We aim to ensure that all information we maintain is accurate, current and fit for the purpose intended. However it does not constitute legal or other professional advice. Neither the MUCM consortium, or its funders RCUK, nor any of the sources of the information shall be responsible for any errors or omissions, or for the use of or results obtained from the use of this information.

Links to and from this site are for convenience only and do not mean that MUCM endorses or approves them. We cannot guarantee that these links will work all of the time and we have no control over the availability of linked pages. We will strive to maintain them and would be grateful to receive information on any broken links if found in order that we can review them. It is the responsibility of the internet user to make their own decisions about the accuracy, currency, reliability and correctness of information found at sites linked from this website.

8.8 Meta-pages: Software Disclaimer

Information at this site is general information provided as part of the RCUK Manging Uncertainties in Complex Models project. We aim to ensure that all information we maintain is accurate, current and fit for the purpose intended. However it does not constitute legal or other professional advice. Neither the *MUCM* consortium, or its funders RCUK, nor any of the sources of the information shall be responsible for any errors or omissions, or for the use of or results obtained from the use of this information.

The *MUCM* toolkit includes references in various places to software that we believe is available to carry out some of the toolkit procedures. Users of the toolkit should note, however, that software does not form part of the toolkit and such references are for convenience only. The MUCM consortium makes no endorsement or warranties about any software referred to in the toolkit, and is not responsible for the quality, reliability, accuracy or safety of such software. This includes both software produced by consortium members or by others.

8.9 Thread: Analysis of the core model using Gaussian Process methods

8.9.1 Overview

The principal user entry points to the *MUCM* toolkit are the various *threads*, as explained in the *Toolkit Structure*. The main threads give detailed instructions for building and using *emulators* in various contexts.

This thread takes the user through the analysis of the most basic kind of problem, using the fully *Bayesian* approach based on a Gaussian process (GP) emulator. We characterise a core problem or model as follows:

- We are only concerned with one *simulator*.
- The simulator only produces one output, or (more realistically) we are only interested in one output.
- The output is *deterministic*.
- We do not have observations of the real world process against which to compare the simulator.
- We do not wish to make statements about the real world process.
- We cannot directly observe derivatives of the simulator.

Each of these aspects of the core problem is discussed further in page *DiscCore*.

The fully Bayesian approach has a further restriction:

- We are prepared to represent the simulator as a *Gaussian process*.

See also the discussion page on the Gaussian assumption (*DiscGaussianAssumption*).

This thread comprises a number of key stages in developing and using the emulator.

8.9.2 Active inputs

Before beginning to develop the emulator, it is necessary to decide what inputs to the simulator will be varied. Complex simulators often have many inputs and many outputs. In the core problem, only one output is of interest and we assume that this has already been identified. It may also be necessary to restrict the number of inputs that will be represented in the emulator. The distinction between *active inputs* and *inactive inputs* is considered in the discussion page *DiscActiveInputs*.

Once the active inputs have been determined, we will refer to these simply as the inputs, and we denote the number of (active) inputs by p.

8.9.3 The GP model

The first stage in building the emulator is to model the mean and covariance structures of the Gaussian process that is to represent the simulator. As explained in the definition of a Gaussian process (*DefGP*), a GP is characterised by a mean function and a covariance function. We model these functions to represent prior beliefs that we have about the simulator, i.e. beliefs about the simulator prior to incorporating information from the *training sample*.

The choice of a mean function is considered in the alternatives page *AltMeanFunction*. In general, the choice will lead to the mean function depending on a set of *hyperparameters* that we will denote by β.

The most common approach is to define the mean function to have the linear form $m(x) = h(x)^{\rm T}\beta$, where $h(\cdot)$ is a vector of regressor functions, whose specification is part of the choice to be made. For appropriate ways to model the mean, both generally and in linear form, see *AltMeanFunction*.

The GP covariance function is discussed in page *DiscCovarianceFunction*. Within the toolkit we will assume that the covariance function takes the form $\sigma^2 c(\cdot,\cdot)$, where σ^2 is an unknown scale hyperparameter and $c(\cdot,\cdot)$ is called the correlation function indexed by a set of correlation hyperparameters δ. The choice of the emulator prior correlation function is considered in the alternatives page *AltCorrelationFunction*.

The most common approach is to define the correlation function to have the Gaussian form $c(x,x') = \exp\{-(x - x')^{\rm T}C(x-x')\}$, where C is a diagonal matrix with elements the inverse squares of the elements of the δ vector. A slightly more complex form is the Gaussian with nugget, $c(x,x') = \nu I_{x=x'} + (1-\nu)\exp\{-(x-x')^{\rm T}C(x-x')\}$, where the *nugget* ν may represent effects of inactive variables and the expression $I_{x=x'}$ takes the value 1 if $x = x'$ and otherwise is 0. See *AltCorrelationFunction* for more details.

The techniques that follow in this thread will be expressed as far as possible in terms of the general forms of the mean and covariance functions, depending on general hyperparameters β, σ^2 and δ. However, in many cases, simpler formulae and methods can be developed when the linear and Gaussian forms are chosen, and some techniques in this thread may only be available in the special cases.

8.9.4 Prior distributions

The GP modelling stage will have described the mean and covariance structures in terms of some hyperparameters. A fully Bayesian approach now requires that we express probability distributions for these that are again *prior* distributions. Possible forms of prior distribution are discussed in the alternatives page on prior distributions for GP hyperparameters (*AltGPPriors*). The result is in general a joint distribution $\pi(\beta, \sigma^2, \delta)$. Where required, we will denote the marginal distribution of δ by $\pi_\delta(\cdot)$, and similarly for marginal distributions of other groups of hyperparameters.

8.9.5 Design

The next step is to create a *design*, which consists of a set of points in the input space at which the simulator is to be run to create the training sample. Design options for the core problem are discussed in the alternatives page on training sample design for the core problem (*AltCoreDesign*).

The result of applying one of the design procedures described there is an ordered set of points $D = \{x_1, x_2, \dots, x_n\}$. The simulator $f(\cdot)$ is then run at each of these input configurations, producing a vector $f(D)$ of n elements, whose i-th element $f(x_i)$ is the output produced by the simulator from the run with inputs x_i.

One suggestion that is commonly made for the choice of the sample size n is $n = 10p$, where p is the number of inputs. (This may typically be enough to obtain an initial fit, but additional simulator runs are likely to be needed for the purposes of *validation*, and then to address problems raised in the validation diagnostics as discussed below.)

8.9.6 Fitting the emulator

Given the training sample and the GP prior model, the procedure for building a GP emulator for the core problem is theoretically straightforward, and is set out in page *ProcBuildCoreGP*. Nevertheless, there are several computational difficulties that are discussed there.

The result of *ProcBuildCoreGP* is the emulator, fitted to the prior information and training data. As discussed fully in the page on forms of GP based emulators (*DiscGPBasedEmulator*), the emulator has two parts, an updated GP (or a related process called a *t-process*) conditional on hyperparameters, plus one or more sets of representative values of those hyperparameters. Addressing the tasks below will then consist of computing solutions for each set of hyperparameter values (using the GP or t-process) and then an appropriate form of averaging of the resulting solutions.

Although the fitted emulator will correctly represent the information in the training data, it is always important to validate it against additional simulator runs. The procedure for validating a Gaussian process emulator is described in page *ProcValidateCoreGP*. It is often necessary, in response to the validation diagnostics, to rebuild the emulator using additional training runs.

8.9.7 Tasks

Having obtained a working emulator, the MUCM methodology now enables efficient analysis of a number of tasks that regularly face users of simulators.

Prediction

The simplest of these tasks is to use the emulator as a fast surrogate for the simulator, i.e. to predict what output the simulator would produce if run at a new point in the input space. The procedure for predicting the simulator's output in one or more new points is set out in page *ProcPredictGP*.

For some of the tasks considered below, we require to predict the output not at a set of discrete points, but in effect the entire output function as the inputs vary over some range. This can be achieved also using simulation, as discussed in the procedure page for simulating realisations of an emulator (*ProcSimulationBasedInference*).

Uncertainty analysis

Uncertainty analysis is the process of predicting the simulator output when one or more of the inputs are uncertain. The procedure page for performing uncertainty analysis using a GP emulator (*ProcUAGP*) explains how this is done.

Sensitivity analysis

In *sensitivity analysis* the objective is to understand how the output responds to changes in individual inputs or groups of inputs. The procedure page for variance based sensitivity analysis using a GP emulator (*ProcVarSAGP*) gives details of carrying out *variance based* sensitivity analysis.

8.9.8 Examples

- *One dimensional example*
- *Two dimensional example* with uncertainty and sensitivity analysis

8.9.9 Additional Comments, References, and Links

Other tasks that can be addressed include optimisation (finding the values of one or more inputs that will minimise or maximise the output) and decision analysis (finding an optimal decision according to a formal description of utilities). A related task is *decision-based* sensitivity analysis. We expect to add procedures for these tasks for the core problem in due course.

Another task that is very often required is *calibration*. This requires us to think about the relationship between the simulator and reality, which is dealt with in *ThreadVariantModelDiscrepancy*. Tasks involving observations of the real process are explicitly excluded from the core problem.

8.10 Thread: Bayes linear emulation for the core model

8.10.1 Overview

This page takes the user through the construction and analysis of an emulator for a simple univariate computer simulator – the *core* problem. The approach described here employs *Bayes linear methods*.

8.10.2 Requirements

The method and techniques described in this page are applicable when we satisfy the following requirements:

- We are considering a core problem with the following features:
  - We are only concerned with one *simulator*.
  - The simulator only produces one output, or (more realistically) we are only interested in one output.
  - The output is *deterministic*.

  - We do not have observations of the real world process against which to compare the simulator.
  - We do not wish to make statements about the real world process.
  - We cannot directly observe derivatives of the simulator.
- We are prepared to represent our beliefs about the simulator with a *second-order specification* and so are following the *Bayes linear* approach

8.10.3 The Bayes linear emulator

The Bayes linear approach to emulation is (comparatively) simple in terms of belief specification and analysis, requiring only mean, variance and covariance specifications for the uncertain output of the computer model rather than a full joint probability distribution for the entire collection of uncertain computer model output. For a detailed discussion of Bayes linear methods, see *DiscBayesLinearTheory*; for discussion of Bayes linear methods in comparison to the Gaussian process approach to emulation, see *AltGPorBLEmulator*.

Our belief specification for the univariate deterministic simulator is given by the Bayes linear *emulator* of the simulator $f(x)$ which takes the a linear *mean function* in the following structural form:

$$f(x) = \sum_j \beta_j \, h_j(x) + w(x)$$

In this formulation, $\beta = (\beta_1, \ldots, \beta_p)$ are unknown scalars, $h(x) = (h_1(x), \ldots, h_p(x))$ are known deterministic functions of x, and $w(x)$ is a stochastic residual process. Thus our mean function has the linear form $m(x) = h(x)^T\beta$.

Thus our belief specification for the computer model can be expressed in terms of beliefs about two components. The component $h^T(x)\beta$ is a linear trend term that expresses our beliefs about the global variation in f, namely that portion of the variation in $f(x)$ which we can resolve without having to make evaluations for f at input choices which are near to x. The residual $w(x)$ expresses local variation, which we take to be a weakly stationary stochastic process with constant variance σ^2 (for a discussion on the covariance function see *DiscCovarianceFunction*), and a specified *correlation function* $c(x, x')$ which is parametrised by correlation hyperparameters δ. We treat β and $w(x)$ as being uncorrelated a priori. The advantages of including a structured mean function, such as the linear form used here, are discussed in *DiscStructuredMeanFunction*.

8.10.4 Emulator prior specification

Given the emulator structure described above, in order to construct a Bayes linear emulator for a given simulator $f(x)$ we require the following ingredients:

- The form of the trend *basis functions* $h(x)$
- Expectations, variances, and covariances for the trend coefficients β
- Expectation of the residual process $w(x)$ at a given input x
- The form of the residual covariance function $c(x, x')$
- One of:
  1. Specified values for the residual variance σ^2 and correlation hyperparameters δ,
  2. Expectations, variances and covariances for (σ^2, δ)
  3. A sufficiently large number of model evaluations to estimate (σ^2, δ) empirically

These specifications are used to represent our prior beliefs that we have about the simulator before incorporating information from the *training sample*. We now discuss obtaining appropriate specifications for each of these quantities.

Choosing the form of $h(x)$

For Bayes linear emulation, the emphasis of the emulator is often placed on a *detailed structural representation* of the simulator's mean behaviour. Therefore the choice of trend basis function is a key component of the BL emulator. This choice can be made directly by an expert or by empirical investigation of a large sample of simulator evaluations. Methods for determining appropriate choices of $h(x)$ are discussed in the alternatives page on basis functions for the emulator mean (*AltBasisFunctions*).

Choosing the form of $c(x, x')$

If the simulator's behaviour is well-captured by the chosen mean function, then the proportion of variation in the simulator output that is explained by the residual stochastic process is quite small making the choice of the form for $c(x, x')$ less influential in subsequent analyses. Nonetheless, alternatives on the emulator prior correlation function are considered in *AltCorrelationFunction*. A typical choice is the Gaussian correlation function for the residuals.

If we have chosen to work with *active inputs* in the mean function, then the covariance function often includes a *nugget* term, representing the variation in the output of the simulator which is not explained by the active inputs. See the discussion page on active and inactive inputs (*DiscActiveInputs*).

Belief specifications for β, σ^2, and δ

The emulator modelling stage will have described the form of the mean and covariance structures in terms of some hyperparameters. A Bayes linear approach now requires that we express our prior beliefs about these hyperparameters.

Given the specified trend functions $h(x)$, we now require an expectation and variance for each coefficient β_j and a covariance between every pair (β_j, β_k). We additionally require a specification of values for the residual variance σ^2 and the correlation function parameters δ. Depending on the availability of expert information and the level of detail of the specification, this may take the form of (a) expert-specified point values, (b) expert-specified expectations and variances, (c) empirically obtained numerical estimates.

As with the basis functions, these specifications can either be made from expert judgement or via data analysis when there are sufficient simulator evaluations. Further details on making these specifications are described in the alternatives page on prior specification for BL hyperparameters (*AltBLPriors*).

8.10.5 Design

The next step is to create a *design*, which consists of a set of points in the input space at which the simulator is to be run to create the training sample. Alternative choices on training sample design for the core problem are given in *AltCoreDesign*.

The result of applying one of the design procedures described there is a matrix of n points $X = (x_1, \dots, x_n)^T$. The simulator is then run at each of these input configurations, producing an n-vector $f(X)$ of elements, whose i-th element is the output $f(x_i)$ produced by the simulator from the run with inputs x_i.

8.10.6 Building the emulator

Empirical construction from runs only

If the prior information is weak and the amount of available data is large, then any Bayesian posterior would be dominated by the data. Thus given a specified form for the simulator mean function, we can estimate β and σ^2 via standard regression techniques. This will give estimates $\hat{\beta}$ and $\hat{\sigma}^2$ which can be treated as adjusted/posterior values for those parameters given the data. The procedure for the empirical construction of a Bayes linear emulator is described in *ProcBuildCoreBLEmpirical*.

Bayes linear assessment of the emulator

Given the output $f(X)$, we make a Bayes linear adjustment of the trend coefficients β and the residual function $w(x)$. This adjustment requires the specification of a prior mean and variance β, a covariance specification for $w(x)$, and specified values for σ^2 and δ. Given the design, model runs and the prior BL emulator the process of *adjusting* β and $w(x)$ is described in the procedure page for building a BL emulator for the core problem (*ProcBuildCoreBL*).

Bayes linear adjustment for residual variance and correlation functions

Before carrying out the Bayes linear assessment as described above, we may learn about the residual variance via Bayes linear *variance learning*. Consequently, we additionally require a second-order prior specification for σ^2 which may come from expert elicitation or analysis of fast approximate models. The procedure for adjusting our beliefs about the emulator residual variance is described in *ProcBLVarianceLearning*.

We may similarly use Bayes linear variance learning methods for updating our beliefs about the correlation function (and hence δ.)

Bayes linear emulator construction with uncertain variance and correlation hyperparameters will be developed in a later version of the Toolkit.

8.10.7 Diagnostics and validation

Although the fitted emulator will correctly represent the information in the simulator runs, it is always important to validate it against additional model evaluations runs. We assess this by applying the diagnostic checks and, if necessary, rebuilding the emulator using runs from an additional design.

The procedure page on validating a Gaussian process emulator (*ProcValidateCoreGP*) describes diagnostics and validation for GP emulators. This approach is generally applicable to the BL case and so can be used to validate a Bayes linear emulator. However unlike the GP diagnostic process, the Bayes linear approach would not consider the diagnostic values to have particular distribution forms. Specific Bayes linear diagnostics will be developed in a future version.

8.10.8 Post-emulation tasks

Having obtained a working emulator, the MUCM methodology now enables efficient analysis of a number of tasks that regularly face users of simulators.

Prediction

The simplest of these tasks is to use the emulator as a fast surrogate for the simulator, i.e. to predict what output the simulator would produce if run at a new point x in the input space. The procedure for predicting one or more new points using a BL emulator is set out in *ProcBLPredict*.

Uncertainty analysis

Uncertainty analysis is the process of predicting the computer model output, when the inputs to the computer model are also uncertain, thereby exposing the uncertainty in model outputs that is attributable to uncertainty in the inputs. The Bayes linear approach to such a prediction problem is described in the procedure page on Uncertainty analysis for a Bayes linear emulator (*ProcUABL*).

Sensitivity analysis

In *sensitivity analysis* the objective is to understand how the output responds to changes in individual inputs or groups of inputs. In general, when the mean function of the emulator accounts for a large proportion of the variation of the simulator then the sensitivity of the simulator to changes in the inputs can be investigated by examination of the basis functions of $m(x)$ and their corresponding coefficients. In the case where the mean function does not explain much of the simulator variation and the covariance function is Gaussian then the methods of the procedure page on variance based sensitivity analysis (*ProcVarSAGP*) are broadly applicable if we are willing to ascribe a prior distributional form to the simulator input.

8.11 Thread: Generic methods to emulate derivatives

8.11.1 Overview

This thread describes how we can build an *emulator* with which we can predict the derivatives of the model output with respect to the inputs. If we have derivative information available, either from an *adjoint* model or some other means, we can include that information when emulating derivatives. This is similar to the variant thread on emulators with derivative information (*ThreadVariantWithDerivatives*) which includes derivative information when emulating function output. If the adjoint to a simulator doesn't exist and we don't wish to obtain derivative information through another method, it is still possible to emulate model derivatives with just the function output.

8.11.2 The emulator

The derivatives of a posterior Gaussian process remain Gaussian processes with mean and covariance functions obtained by the relevant derivatives of the posterior mean and covariance functions. This can be applied to any Gaussian process emulator. The process of building an emulator of derivatives with the fully Bayesian approach is given in the procedure page *ProcBuildEmulateDerivsGP*. This covers building a Gaussian process emulator of derivatives with just function output, an extension of the core thread *ThreadCoreGP*, and a Gaussian process emulator of derivatives built with function output and derivative information, an extension of *ThreadVariantWithDerivatives*.

The result is a Gaussian process emulator of derivatives which will correctly represent any derivatives in the training data, but it is always important to validate the emulator against additional derivative information. For the *core problem*, the process of validation is described in the procedure page *ProcValidateCoreGP*. Although here we are interested in emulating derivatives, as we know the derivatives of a Gaussian process remain a Gaussian process, we can apply the same validation techniques as for the core problem. We require a validation design D' which consists of points where we want to obtain validation derivatives. An *adjoint* is then run at these points; if an appropriate adjoint does not exist the derivatives are obtained through another technique, for example finite differences. If any local sensitivity analysis has already been performed on the simulator, some derivatives may already have been obtained and can be used here for validation. Then in the case of a linear mean function, weak prior information on hyperparameters β and σ, and a single posterior estimate of δ, the predictive mean vector, m^*, and the predictive covariance matrix, V^*, required in *ProcValidateCoreGP*, are given by the functions $\tilde{m}^*(\cdot)$ and $\tilde{v}^*(\cdot,\cdot)$ which are given in *ProcBuildEmulateDerivsGP*. We can therefore validate an emulator of derivatives using the same procedure as that which we apply to validate an emulator of the core problem. It is often necessary, in response to the validation diagnostics, to rebuild the emulator using additional training runs which can of course, include derivatives. We hope to extend the validation process using derivatives as we gain more experience in validation diagnostics and emulating with derivative information.

The *Bayes linear* approach to emulating derivatives may be covered in a future release of the toolkit.

8.11.3 Tasks

Having obtained a working emulator, the MUCM methodology now enables efficient analysis of a number of tasks that regularly face users of simulators.

Prediction

The simplest of these tasks is to use the emulator as a fast surrogate for the adjoint, i.e. to predict what derivatives the adjoint would produce if run at a new point in the input space. The process of predicting function output at one or more new points for the core problem is set out in the prediction page *ProcPredictGP*. Here we are predicting derivatives and the process of prediction is the same as for the core problem. If the procedure in *ProcBuildEmulateDerivsGP* is followed, $\tilde{D}, \tilde{t}, \tilde{A}, \tilde{e}$ etc. are used in replace of D, t, A, e, as required in *ProcPredictGP*.

Sensitivity analysis

In *sensitivity analysis* the objective is to understand how the output responds to changes in individual inputs or groups of inputs. Local sensitivity analysis uses derivatives to study the effect on the output, when the inputs are perturbed by a small amount. Emulated derivatives could replace adjoint produced derivatives in this analysis if the adjoint is too expensive to execute or in fact does not exist.

Other tasks

Derivatives can be informative in optimization problems. If we want to find which sets of input values results in either a maximum or a minimum output then knowledge of the gradient of the function, with respect to the inputs, may result in a more efficient search. Derivative information is also useful in *data assimilation*.

8.12 Thread: History Matching

8.12.1 Overview

The principal user entry points to the MUCM toolkit are the various threads, as explained in *MetaToolkitStructure*. This thread takes the user through a technique known as *history matching*, which is used to learn about the inputs x to a model $f(x)$ using observations of the real system z. As the history matching process typically involves the use of expectations and variances of *emulators*, we assume that the user has successfully emulated the model using the Bayes Linear strategy as detailed in *ThreadCoreBL*. An associated technique corresponding to a fully probabilistic emulator, as described in *ThreadCoreGP*, will be discussed in a future release. Here we use the term model synonymously with the term *simulator*.

The description of the link between the model and the real system is vital in the history matching process, therefore several of the concepts discussed in *ThreadVariantModelDiscrepancy* will be used here.

As we are not concerned with the details of emulation we will describe a substantially more general case than covered by the *core model*. Assumptions we share with the core model are:

- We are only concerned with one simulator.
- The simulator is deterministic.

However, in contrast to the Core model we now assume:

- We have observations of the real world process against which to compare the simulator.
- We wish to make statements about the real world process.

- The simulator can produce one, or more than one, output of interest.

The first two of these new assumptions are fundamental to this thread as we will be comparing model outputs with real world measurements. We then use this information to inform us about the model inputs x. The third point states that we will be dealing with both univariate and multivariate cases. In this release we discuss the *Bayes Linear case*, where our beliefs about the simulator are represented as a *second-order specification*. Other assumptions such as whether simulator derivative information is available (which could have been used in the emulation construction process), do not concern us here.

8.12.2 Notation

In accordance with standard *toolkit notation*, in this page we use the following definitions:

- x - vector of inputs to the model
- $f(x)$ - vector of outputs of the model function
- y - vector of the actual system values
- z - vector of observations of reality y
- x^+ - 'best input'
- d - model discrepancy
- $\mathcal{X}$ - set of all acceptable inputs
- $\mathcal{X}_0$ - whole input space
- $\mathcal{X}_j$ - reduced input space after j waves
- $I(x)$ - implausibility measure

8.12.3 Motivation for History Matching

It is often the case that when dealing with a particular model of a physical process, observations of the real world system are available. These observations z can be compared with outputs of the model $f(x)$, and often a major question of interest is: what can we learn about the inputs x using the observations z and knowledge of the model $f(x)$.

History matching is a technique which seeks to identify regions of the input space that would give rise to acceptable matches between model output and observed data, a set of inputs that we denote as $\mathcal{X}$. Often large parts of the input space give rise to model outputs that are very different from the observed data. The strategy, as is described below, involves iteratively discarding such 'implausible' inputs from further analysis, using straightforward, intuitive criteria.

At each iteration this process involves: the construction of emulators (which we will not discuss in detail here); the formulation of *implausibility measures* $I(x)$; the imposing of cutoffs on the implausibility measures, and the subsequent discarding of unwanted (or implausible) regions of input space.

Often in computer model experiments, the vast majority (or even all) of the input space would give rise to unacceptable matches to the observed data, and it is these regions that the history matching process seeks to identify and discard. Analysis of the often extremely small volume that remains can be of major interest to the modeller. This might involve analysing in which parts of the space acceptable matches can be found, what are the dependencies between acceptable inputs and what is the quality of matches that are possible. The goal here is just to rule out the obviously bad parts: for a more detailed approach involving priors and posterior distributions for the best input x^+, the process known as calibration has been developed. This will be described in a future release, including a comparison between the calibration and history matching processes.

8.12.4 Implausibility Measures

The history matching approach is centred around the concept of an *implausibility measure* which we now introduce, for further discussion see *AltImplausibilityMeasure*. An implausibility measure $I(x)$ is a function defined over the whole input space which, when large, suggests that there would be a large disparity between the model output and the observed data.

We do not know the model outputs $f(x)$ corresponding to every point x in input space, as the model typically takes too long to run. In order to construct such an implausibility measure, we first build an emulator (such as is described in the *core Bayes Linear thread*) in order to obtain the expectation and variance of $f(x)$. We then compare the expected output $\mathrm{E}[f(x)]$ with the observations z. In the simplest case where $f(x)$ represents a single output and z a single observation, a possible form for the univariate implausibility measure is:

$$I^2(x) = \frac{(\mathrm{E}[f(x)] - z)^2}{\mathrm{Var}[\mathrm{E}[f(x)] - z]} = \frac{(\mathrm{E}[f(x)] - z)^2}{\mathrm{Var}[f(x)] + \mathrm{Var}[d] + \mathrm{Var}[e]}$$

where $\mathrm{E}[f(x)]$ and $\mathrm{Var}[f(x)]$ are the emulator expectation and variance respectively, d is the *model discrepancy*, discussed in *ThreadVariantModelDiscrepancy* and e is the observational error. The second equality follows from the definition of the *best input* approach (see *DiscBestInput* for details).

The basic idea is that if $I(x)$ is high for some x, then even given all the uncertainties present in the problem, we would still expect the output of the model to be a poor match to the observed data z. We can hence discard x as a potential member of the set $\mathcal{X}$.

As is discussed in *AltImplausibilityMeasure*, there are many possible choices of measure. If the function has many outputs then one can define a univariate implausibility measure $I_{(i)}(x)$ for each of the outputs labelled by i. One can then use the maximum implausibility $I_M(x)$ to discard input space. It is also possible to define a full multivariate implausibility measure $I_{MV}(x)$, provided one has available suitable multivariate versions of the model discrepancy d (see for example *DiscStructuredMD*), the observational errors, and a multivariate emulator. (A multivariate emulator is not essential if, for example, the user has an accurate multi-output emulator: see *ThreadVariantMultipleOutputs*).

Implausibility measure are simple and intuitive, and are easily constructed and used, as is described in the next section.

8.12.5 Imposing Implausibility Cutoffs

The history matching process seeks to identify the set of all inputs $\mathcal{X}$ that would give rise to acceptable matches between outputs and observed data. Rather that focus on identifying such acceptable inputs, we instead discard inputs that are highly unlikely to be members of $\mathcal{X}$.

This is achieved by imposing cutoffs on the implausibility measures. For example, if we were dealing with the univariate implausibility measure defined above, we might impose the cutoff c and discard from further analysis all inputs that do not satisfy the constraint $I(x) \leq c$ This defines a new sub-volume of the input space that we refer to as the non-implausible volume, and denote as $\mathcal{X}_1$. The choice of value for the cutoff c is obviously important, and various arguments can be employed to determine sensible values, as are discussed in *DiscImplausibilityCutoff*. A common method is to use Pukelsheim's (1994) three-sigma rule that states that for any unimodal, continuous distribution 0.95 of the probability will lie within a $\pm 3\sigma$ interval. This suggests that taking a value of $c = 3$ is a reasonable starting point for a univariate measure.

Suitable cutoffs for each of the implausibility measures introduced in *AltImplausibilityMeasure*, such as $I_M(x)$ and $I_{MV}(x)$, can be found through similar considerations. This often involves analysing the fraction of input space that would be removed for various sizes of cutoff (see *DiscImplausibilityCutoff*). In many applications, large amounts of input space can be removed using relatively conservative (i.e. large) choices of the cutoffs.

We apply such space reduction steps iteratively, as described in the next section.

8.12.6 Iterative Approach to Input Space Reduction

As opposed to attempting to identify the set of acceptable inputs $\mathcal{X}$ in one step, we instead employ an iterative approach to input space reduction. At each iteration or *wave*, we design a set of runs only over the current non-implausible volume, emulate using these runs, calculate the implausibility measures of interest and impose cutoffs to define a new (smaller) non-implausible volume. This is referred to as refocusing.

The full iterative method can be summarised by the following algorithm. At each iteration or wave:

1. A design for a space filling set of runs over the current non-implausible volume $\mathcal{X}_j$ is created.
2. These runs (along with any non-implausible runs from previous waves) are used to construct a more accurate emulator defined only over the current non-implausible volume $\mathcal{X}_j$.
3. The implausibility measures are then recalculated over $\mathcal{X}_j$, using the new emulator,
4. Cutoffs are imposed on the implausibility measures and this defines a new, smaller, non-implausible volume $\mathcal{X}_{j+1}$ which should satisfy $\mathcal{X} \subset \mathcal{X}_{j+1} \subset \mathcal{X}_j$.
5. Unless the stopping criteria described below have been reached, or the computational resources exhausted, return to step 1.

At each wave the emulators become more accurate, and this allows further reduction of the input space. Assuming sufficient computational resources, the stopping criteria are achieved when, after a (usually small) number of waves, the emulator variance becomes far smaller than the other uncertainties present, namely the model discrepancy and observational errors. At this point the algorithm is terminated. The current non-implausible volume $\mathcal{X}_j$ should be a reasonable approximation to the acceptable set of inputs $\mathcal{X}$. For further details and discussion of why this method works, and for full descriptions of the stopping criteria, see *DiscIterativeRefocussing*.

8.12.7 A 1D Example

For a simple, illustrative example of the iterative approach to history matching, see *Exam1DHistoryMatch* where a simple 1-dimensional model is matched to observed data using two waves of refocussing.

8.12.8 Additional Comments, References and Links.

While the goal of this approach is to identify the set of acceptable inputs $\mathcal{X}$, it is possible that this set is empty. This possibility would be identified by the history matching approach, and we would therefore suggest that history matching is employed first, before other more detailed techniques are used. Once it is established that the set $\mathcal{X}$ is non-empty, and once the location, size and structure of $\mathcal{X}$ have been analysed (which are often of major interest to the modellers), then more detailed techniques such as probabilistic calibration can be employed.

Note that if, at any wave we find that the set $\mathcal{X}_k$ is empty, then we would declare that $\mathcal{X}$ is empty also, and therefore that the simulator does not provide acceptable matches to the observed data. Conversely, we can establish that $\mathcal{X}$ is non-empty by checking to see if any of the runs we have used, in any of the waves, would pass all the implausibility cutoffs. If so these runs are, by definition, members of $\mathcal{X}$. If we have reached the stopping criteria after k iterations and have not found any such runs, we can do a final batch of runs provided $\mathcal{X}_k$ is still non-empty.

The iterative refocussing strategy presented in this thread is a very powerful method and has been successfully used to history match complex models across a variety of application areas. These include oil reservoir models (Craig et. al. 1996, 1997) and models of Galaxy formation (Vernon et. al. 2010, Bower et. al. 2009).

Pukelsheim, F. (1994). “The three sigma rule.” *The American Statistician*, 48: 88–91.

Craig, P. S., Goldstein, M., Seheult, A. H., and Smith, J. A. (1996). “Bayes linear strategies for history matching of hydrocarbon reservoirs.” In Bernardo, J. M., Berger, J. O., Dawid, A. P., and Smith, A. F. M. (eds.), Bayesian Statistics 5, 69–95. Oxford, UK: Clarendon Press.

Craig, P. S., Goldstein, M., Seheult, A. H., and Smith, J. A. (1997). "Pressure matching for hydrocarbon reservoirs: a case study in the use of Bayes linear strategies for large computer experiments." In Gatsonis, C., Hodges, J. S., Kass, R. E., McCulloch?, R., Rossi, P., and Singpurwalla, N. D. (eds.), Case Studies in Bayesian Statistics, volume 3, 36–93. New York: Springer-Verlag.

Vernon, I., Goldstein, M., and Bower, R. (2010), "Galaxy Formation: a Bayesian Uncertainty Analysis," MUCM Technical Report 10/03

Bower, R., Vernon, I., Goldstein, M., et al. (2009), "The Parameter Space of Galaxy Formation," to appear in MNRAS; MUCM Technical Report 10/02

8.13 Thread: Generic methods for combining two or more emulators

8.13.1 Overview

In this thread we consider the situation in which two or more independent *emulators* have been built. Each emulates a single output of a *simulator*, and we assume that they all have the same set of inputs. This situation will usually arise when the emulators are emulating different outputs from the same simulator - see the alternatives page on approaches to emulating multiple outputs (*AltMultipleOutputsApproach*). It may also arise when they are emulating outputs from different simulators, and here the requirement for all the emulators to have the same set of inputs may be met by assembling all the inputs for the different simulators into a single combined set (so that an individual emulator formally has this pooled set as its inputs although it only depends on a subset of them).

The emulators have been built separately and are supposed to be independent. Our objective in this thread is to combine the emulators to address tasks which do not relate to a single output. Here are some examples.

- A simulator of an engine outputs the fuel consumption in each minute during a loading test. In order to estimate consumption at different time points we have considered the alternative emulation approaches discussed in *AltMultipleOutputsApproach* and decided to build separate, independent emulators. However, we are also interested in the cumulative consumption at the end of the first hour, which is the sum of the separate minute-by-minute consumption outputs.
- A simulator outputs the average electricity demand per capita in a city on a given day. Its inputs include weather conditions and demographic data about the city. Separate emulators are built for a number of different cities, in which the demographic data for each city are fixed and the emulator takes just the weather variables as inputs. (It is common to use emulation in this way, to emulate the simulator output for a specific application in which the simulator inputs specific to that application are fixed.) We now wish to estimate total electricity demand over these cities on that day, which is the sum of the various per capita demands weighted by the city populations. Note that although we may be able to assume that the temperature is the same in all the cities on that day, the rainfall and cloud cover may vary between cities, so we would need to use the device of pooling inputs.

This thread therefore deals with how to combine two or more independent emulators to address such questions.

8.13.2 The emulators

We consider emulators built using any of the methods described in other threads, for instance the core threads *ThreadCoreGP* and *ThreadCoreBL*. Therefore they may be fully *Bayesian Gaussian process* (GP) emulators or *Bayes linear* (BL) emulators. The two forms of emulator are different, so this thread addresses separate issues for combining GP or BL emulators where appropriate.

- A GP emulator is in the form of an updated GP (or a related process called a *t-process*) conditional on hyperparameters, plus one or more sets of representative values of those hyperparameters.
- A BL emulator consists of a set of updated means, variances and covariances for the simulator output.

We suppose therefore that we are interested in one or more functions of the simulator outputs. These functions may be linear, as in the above examples, or nonlinear.

We also suppose that the emulators have been *validated* as described in the relevant thread.

8.13.3 Tasks

Having obtained a working emulator, the MUCM methodology now enables efficient analysis of a number of tasks that regularly face users of simulators. In these tasks, the procedure may simply reduce to combining the results of performing the corresponding task on each emulator separately; however, some tasks require additional computations.

Prediction

The simplest task for an emulator is as a fast surrogate for the simulator, i.e. to predict what output the simulator would produce if run at a new point in the input space. Procedures for predicting one or more functions of independently emulated outputs are set out in the procedure page *ProcPredictMultipleEmulators*.

Uncertainty analysis

Uncertainty analysis is the process of predicting the simulator output when one or more of the inputs are uncertain. The procedure page *ProcUAMultipleEmulators* explains how this is done for functions of independently emulated outputs.

8.13.4 Additional Comments, References, and Links

Other tasks that can be addressed include sensitivity analysis (studying how outputs are influenced by individual inputs), optimisation (finding the values of one or more inputs that will minimise or maximise the output) and decision analysis (finding an optimal decision according to a formal description of utilities). We expect to add procedures for these tasks for multiple emulators in due course.

8.14 Thread: Experimental design

This thread presents different ways of an selecting experimental *design*, namely a set of input combinations at which to make computer runs, to construct an *emulator*. We may use the term "design space" to mean the region of input space from which the combinations may be selected and the term "design point" for an actual combination. Designs separate into two basic classes: general purpose designs, which may be used for a range of different *simulators*, and designs which are chosen to be optimal (in some sense) for a particular simulator.

For computer experiments to build an emulator the most used general purpose designs are *space filling designs*. *model based designs* which are based on the principles of optimal design but tailor-made for computer experiments.

Solving optimality problems requires optimisers, that is optimisation algorithms. Exchange algorithms (*ProcExchangeAlgorithm*) are favoured in which, naively, "bad" design points are exchanged for "good". They are cheap to implement and very fast. They are similar in style to global optimisation algorithms, particularly those versions which include a random search element.

A special technique to approximate the process and its covariance function, which provides a useful simplification, is the Karhunen-Loeve expansion: *DiscKarhunenLoeveExpansion* and this can be adapted to experimental design.

An ideal is to be able to conduct sequential experiments in which one can adapt present computer runs to data and analysis conducted previously. Some methods are being developed in *ProcASCM*.

8.14.1 Overview

In any scientific study experimental design should play an important part. The terminology changes, somewhat, between areas and even within statistics itself. Similar terms are: planning of experiments, design of a training set, supervised learning, selective sampling, spatial sampling and so on. Historically, “experiment” has always been a key part of the scientific method. Francis Bacon (1620, Novum Organum):

> *. . . the real order of experience begins by setting up a light, and then shows the road by it, commencing with a regulated and digested, not a misplaced and vague course of experiment, and hence deducing axioms, and from those axioms new experiments. . .*

The subject came of age with the introduction of properly planned agricultural field trials in the 1920s by R A Fisher. It then developed into combinatorial design methods (Latin Squares etc), *factorial design*, response surface design, *optimal design* both classical and Bayesian.

In this thread we focus on methods of computer experimental design for the purpose of building an emulator.

8.14.2 Developing a protocol for a computer experiments study

A scientific study requires an experimental protocol, even if the initial protocol is adapted to circumstances. Experiments need to be planned. This is as much true of a computer experiment as of a physical experiment. We divide this section into two subsections, the first covers some essential planning issues, the second gives a simple standard scheme or *protocol*.

Planning

The following are some issues which are important when planning a computer experiment.

1. The objective of the experiment. There are many, e.g. (i) to understand the input-output relationship (ii) to find the most important explanatory factors (iii) to find the inputs which place the output in some target region or achieve a given level (the inverse problem) (iv) to find the input which maximises the output and what the maximum value is.

2. A starting model. (i) What do we know already about the relationship between inputs and outputs? (ii) Can we rank the inputs in perceived order of importance? Are there regions of input space for which the output has highly variable output?

3. Input and output variables. A full description is essential: (i) units of measurement (ii) ranges (iii) discrete, continuous, functional, etc (iv) our ability to set levels, measure, record, store etc. In summary: each variable needs a full “CV”.

4. Input variables (factors) (i) nominal (central) values, (ii) range, (ii) method of setting levels: by hand, automated, input files etc.

5. Experimental costs/resources. (i) run time (physical or computer experiments) (ii) budget etc. A good measure of cost is how many hours, to set up, run the computer, etc to obtain results of a unit of experimental activity.

A simple four-stage protocol

It is unwise to launch a study with one large experiment. The following is a basic protocol. Each stage will need an experimental design and one should only proceed to the next stage after analysing the results of the previous stage. Analysis is only discussed in this thread to the extent needed for design, but it is helpful to provide diagrammatic representations of results e.g. (i) tables of which input affects which output (ii) basic effect plots.

1. *Nominal experiment*. Set all inputs to their nominal values and generate the output(s). This provides a useful check on (i) the performability of a basic run (ii) a central input-to-output combination (iii) data on set-up time, run time, etc. By experimenting at the "centre" of the input space a useful bench-mark for future runs is provided.

2. *Initial screening experiment*. One may use a formal screening design. The purpose is to identify input factors which significantly affect one or more outputs, with a view to not including (or keeping at their nominal values) the non-significant factors. Even keeping all input factors at nominal and moving just one factor of interest is useful, although inefficient as part of a larger experiment.

3. *Main experiment*. This involves the design and conduct of a larger scale experiment making use of (i) perceived significant inputs (ii) prior knowledge of possible models. It is here that a more sophisticated design for computer experiments may be used.

4. *Confirmatory experiment (validation experiment)*. At a basic level it is useful to have additional training runs as an overall check on the accuracy/validity of the emulator. If the experiments are a success they will confirm or disconfirm prior beliefs about relationships, discover new ones, achieve some optimum etc. It is often important to carry out a more focused confirmatory follow-up experiment. For example, if it is considered that a set of input values puts the output in a target region, then confirmatory runs can try to confirm this.

8.14.3 Main experiment design for an emulator

We now consider in some depth the design of the "main experiment" as described above, with which to build an emulator. The set of design points together with the output in this case is commonly referred to as the *training sample*. General discussion on the design of a training sample is given in the page *DiscCoreDesign*, and we provide here some more technical background. We will return briefly to consideration of screening and validation designs in the final section of this thread.

The most widely used training sample designs are general purpose designs, particularly those that have a *space-filling* property. Such designs attempt to place the design points in the design space so that they are well separated and cover the design space evenly. The rationale for such designs rests on the fact that the simulator output is assumed to vary smoothly as the inputs change, and so in the case of a *deterministic* simulator there is very little extra information to be gained by placing two design points very close to each other. Having design points very close together can also lead to numerical difficulties (as discussed in the page *DiscBuildCoreGP*). Conversely, leaving large "holes" in the design space risks missing important local behaviour of the simulator.

General purpose designs have a long history in experimental design and *DiscFactorialDesign* gives a short introduction. One could consider a space-filling design as a very special type of factorial design, again tailored to computer experiments. In the same way that classical factorial designs give a certain amount of robustness against different possible simple polynomial models, so space-filling designs guard against, or prepare for the presence of, different output features that may arise in different parts of the design space.

Such general-purpose designs have been widely and successfully used in computer experiments. But there are several reasons to look at more sophisticated "tailor-made" designs. For instance, not having points close together makes it more difficult to identify the form and parameters of a suitable covariance function (see the discussion of covariance functions in the page *AltCorrelationFunction* and of estimating their parameters in *AltEstimateDelta*). Also, sequential design procedures may allow the main experiment to adapt to information in earlier stages when planning later stages. (Although some non-random space-filling designs presented in the page *AltCoreDesign* may be used in a sequential way, they are not adaptive.) As a result, there is growing interest in *model-based* optimal designs for training samples.

The Bayesian approach is very useful in underpinning the principals of optimal design because it gives well-defined meaning to the increase in precision or information expected from an experiment. It is also natural because in *MUCM*, we choose Bayesian models to build the emulator.

Model based optimal design is critically dependent on the criteria used. One way to think of optimal design is as a special type of decision problem, and like all decision problems some notion of optimality is needed (in economics one would have a utility function whose expectation is a risk function). There are well-known criteria which were first introduced in (non-Bayesian) classical regression analysis but are now fully adapted to the Bayesian setting. An example of a Bayesian principal working is in understanding the expected again in information from an experiment. All these matters are discussed in *AltOptimalCriteria*. Further discussion of basic optimal design for computer experiments can also be found in *AltCoreDesign*.

In the same way that model-based optimal experimental design grew out of a more decision-theoretical approach to factorial design in regression, so optimal design for computer experiments is a second or even third generation approach to experimental design. The methodology behind optimal design for computer experiments remains, here, Bayes optimal design, but two issues (at least) distinguish the emphasis of optimal design for computer experiments from that for Bayes optimal design in regression. The first is that the criteria are most often based on prediction because the overall quality of the emulator fit is important. Second, the covariance parameters appear in the Gaussian Process model in a non-linear way (see *AltCorrelationFunction*), making optimal design for covariance estimation more intractable when the covariance parameters are unknown.

- *Optimisation*. Solving an optimality problem requires and optimisation algorithm. Exchange algorithms (see the procedure page *ProcExchangeAlgorithm*) iterativley swap one or more points in the design for the same number of points in the candidate set, but outside the design, with the aim of exchanging "bad" points for "better" points. The algorithms are simple to implement and fast, but not guaranteed to converge to the globally best solution. More sophisticated algorithms such as branch and bound which give a global optimum (see *ProcBranchAndBoundAlgorithm*) are available but slower and harder to implement.
- *The Karhunen-Loeve expansion*. A promising way to handle the nonlinearity of the covariance function in its parameters is to use the Karhunen-Loeve expansion. This approach is described in more detail below.
- *Sequential design*. We have already mentioned the potential value of sequential design and this is also discussed below.

Karhunen Loeve (K-L) method

This is a method for representing a Gaussian Process and its covariance function as arising from a random regression with an infinite number of regression functions; see *DiscKarhunenLoeveExpansion*. These function are "orthogonal" in a well-defined sense. By truncating the series, and equivalently its covariance function, we obtain an approximation to the process but one which makes it an ordinary random regression and therefore amenable to standard Bayes optimal design methods; see *AltOptimalCriteria*. To use the K-L method one needs to compute the expansion numerically because there are very few cases in which there is a closed form solution. The K-L method is one way of avoiding the problems associated with optimal design for covariance parameters which arise because of the non-linearity. Another benefit is that one can see how the smoothness of the process is split between different terms; typically slowly varying terms lead to design points which are more extreme or concentrate on few areas whereas high frequency terms tend to require designs points which are spread inside the design space.

Sequential experiments

Sequential methods in experimental design can be simple; the above four-stage protocol can be considered as a type of sequential experiment. Full sequential procedures use the data and the analysis from previous experiments to select further experiments. They can be one design point at a time or block sequential. The Bayes paradigm is very useful in understanding sequential experimental design and in *AltOptimalCriteria* there is a discussion. The basic strategy is to update parameter estimates, of both the "regression" and covariance parameters, and base the next design point or block of design points on the updated assessment of the underlying Gaussian process. As mentioned, criteria which depend on prediction quality are favoured.

It is useful to think of sequential design as being partly adaptive in the case where outputs play little or no role in the choice of the next block of designs points and fully adaptive, where both inputs and outputs are used. The partly adaptive material appears in *ProcASCM*. Fully adaptive methods will appear in later releases of the toolkit, using the partly adaptive methods as a foundation.

8.14.4 Design for other toolkit areas

Screening design

Screening was discussed earlier in the context of the second stage of the four-stage protocol. Screening methods, with the resulting specialised designs, are considered in the topic thread *ThreadTopicScreening*.

Validation design

Validation was discussed in the context of the fourth stage of the four-stage protocol. Suitable criteria and designs for validation are an active topic of research and we expect to provide more discussion in a later release of the toolkit. Some interim ideas are presented in *DiscCoreValidationDesign*.

Simulation design

Thi kind of design that arise in the toolkit is in the context of simulation techniques for computing predictions and other more complex tasks from emulators. As discussed in *ProcSimulationBasedInference*, the general simulation method involves drawing simulated realisations of the simulator itself, and the associated design issue is discussed in *DiscRealisationDesign*. This is another area where more research is needed and we hope to report progress in later releases of the toolkit.

Design for combined physical and computer experiments

An outstanding problem is to design experiments which are a mixture of computer experiments (simulation runs) and physical experiments. Some of the issues come under the heading of *calibration*. A simple protocol is to do physical experiments to improve the predictions of constants, features or simply the model itself where these are predicted by the emulator to be poor (high discrepancy) or where the uncertainty is large (high posterior variance). An ideal Bayesian approach is to combine the emulator and real world model into a single modelling system, given a full joint prior distribution. This model-based approach may eventually lead to more coherent optimal design than simple protocols of the kind just mentioned. The importance of this area cannot be underestimated.

8.14.5 Additional Comments, References, and Links

The following books have some design material.

Thomas J. Santner, Brian J. Williams, William Notz. The design and analysis of computer experiments. Springer, 2003

K. Fang, R. Lui and A.Sudjianto. Design and modelling for computer experiments. Chapman and Hall/CRC, 2005.

A recent paper on computer/physical experiments is:

D. Romano (with A Giovagnoli) A sequential methodology for integrating physical and computer experiments. Presentation at the Newton Institute. http://www.newton.ac.uk/programmes/DOE/seminars/081515001.html

8.15 Thread: Screening

8.15.1 Overview

Screening involves identifying the relevant input factors that drive a *simulator's* behaviour.

Screening, also known as variable selection in the machine learning literature, is a research area with a long history. Traditionally, screening has been applied to physical experiments where a number of observations of reality are taken. One of the primary aims is to remove, or reduce, the requirement to measure inconsequential quantities (inputs) thus decreasing the time and expense required for future experiments. More recently, screening methods have been developed for computer experiments where a simulator is developed to model the behaviour of a physical, or other, system. In this context, the quantities represent the input variables and the benefit of reducing the dimension of the input space is on the *emulator* model complexity and training efficiency rather than on the cost of actually obtaining the input values themselves.

With the increasing usage of ever more complex models in science and engineering, dimensionality reduction of both input and output spaces of models has grown in importance. It is typical, for example in complex models, to have several tens or hundreds of input (and potentially output) variables. In such high dimensional spaces, efficient algorithms for dimensionality reduction are of paramount importance to allow effective probabilistic analysis. For very high (say over 1000) sizes of input and/or output spaces open questions remain as to what can be achieved (see for example the variant thread *ThreadVariantMultipleOutputs* for a discussion of the emulation of multiple outputs, and procedure page *ProcOutputsPrincipalComponents* for a description of the use of principal component analysis to reduce the dimension of the output space in particular). Even in simpler models, efficient application of screening methods can reduce the computational cost and permit a focused investigation of the relevant factors for a given model.

Screening is a constrained version of dimensionality reduction where a subset of the original variables is retained. In the general dimensionality reduction case, the variables may be transformed before being used in the emulator, typically using a linear or non-linear mapping.

Both *screening* and *Sensitivity Analysis* (SA) may be utilised to identify variables with negligible total effects on the output variables. They can provide results at various levels of granularity from a simple qualitative ranking of the importance of the input variables through to more exact quantitative results. SA methods provide more accurate variable selection results but require larger number of samples, and thus entail much higher computational cost. The class of SA methods are examined separately, in the topic thread on sensitivity analysis (*ThreadTopicSensitivityAnalysis*).

The focus in this thread is on screening and thus largely qualitative types of methods which typically require lower computational resources, making them applicable to more complex models. Screening methods can be seen as a form of pre-processing and the simulator evaluations used in the screening activity can also be used to construct the emulator.

The benefits of screening are many fold:

1. Emulators are simpler; the reduced input space typically results in simpler models with fewer (hyper)parameters that are more efficient, both to estimate and use.
2. Experimental design is more efficient, in a sequential setting; the initial expense of applying screening is typically more than recouped since a lower dimensional input space can be filled with fewer design points.
3. Interpretability is improved; the input variables are not transformed in any way and thus the practitioner can immediately infer that the quantities represented in the discarded variables need not be estimated or measured in the future.

Screening can be employed as part of the emulator construction and in practice is often applied prior to many of the other methods described in the MUCM toolkit.

In this thread we restrict our attention to single outputs, i.e. for multiple output simulators the outputs would need to be treated independently, and then the active inputs for each output identified separately.

In the alternatives page on screening methods (*AltScreeningChoice*) we provide more details of the alternative approaches to screening that are possible and discuss under what scenarios each screening method may be appropriate.

After the screening task is completed, the identified inactive factors may be excluded from further analysis as detailed in the discussion page on active and inactive inputs (*DiscActiveInputs*).

8.15.2 References

Saltelli, A., Chan, K. and Scott, E. M. (eds.) (2000). Sensitivity Analysis. Wiley.

8.16 Thread: Sensitivity analysis

8.16.1 Overview

This is a topic thread, on the topic of *sensitivity analysis*. Topic threads are designed to provide additional background information on a topic, and to link to places where the topic is developed in the core, variant and generic threads (see the discussion of different kinds of threads in the Toolkit structure page (*MetaToolkitStructure*)).

The various forms of sensitivity analysis (SA) are tools for studying the relationship between a *simulator*'s inputs and outputs. They are widely used to undestand the behaviour of the simulator, to identify which inputs have the strongest influence on outputs and to put a value on learning more about uncertain inputs. It can be a prelude to simplifying the simulator, or to constructing a simplified *emulator*, in which the number of inputs is reduced.

Procedures for carrying out SA using emulators are given in most of the core, variant and generic threads in the toolkit. This topic thread places those specific tools and procedures in the wider context of SA methods and discusses the practical uses of such methods.

8.16.2 Uses of SA

As indicated above, there are several uses for SA. We identify four uses which are outlined here and discussed further when we present particular SA methods.

- *Understanding.* Techniques to show how the output $f(x)$ behaves as we vary one or more of the inputs x are an aid to understanding f. They can act as a 'face validity' check, in the sense that if the simulator is responding in unexpected ways to changes in its inputs, or if some inputs appear to have unexpectedly strong influences, then perhaps there are errors in the mathematical model or in its software implementation.
- *Dimension reduction.* If SA can show that certain inputs have negligible effect on the output, then this offers the prospect of simplifying f by fixing those inputs. Whilst this does not usually simplify the simulator in the sense of making it quicker or easier to evaluate $f(x)$ at any desired x, it reduces the dimensionality of the input space. This makes it easier to understand and use.
- *Analysing output uncertainty.* Where there is uncertainty about inputs, there is uncertainty about the resulting outputs $f(x)$; quantifying the output uncertainty is the role of *uncertainty analysis*, but there is often interest in knowing how much of the overall output uncertainty can be attributed to uncertainty in particular inputs or groups of inputs. In particular, effort to reduce output uncertainty can be expended most efficiently if it is focused on those inputs that are influencing the output most strongly.
- *Analysing decision uncertainty.* More generally, uncertainty about simulator outputs is most relevant when those outputs are to be used in the making of some decision (such as using a climate simulator in setting targets for the reduction of carbon dioxide emissions). Again it is often useful to quantify how much of the overall decision uncertainty is due to uncertainty in particular inputs. The prioritising of research effort to reduce uncertainty depends on the influence of inputs, their current uncertainty and the nature of the decision.

8.16.3 Probabilistic SA

Although a variety of approaches to SA have been discussed and used by people who study and use simulators, there is a strong preference in *MUCM* for a methodology known as probabilistic SA. Other approaches and the reasons for preferring probabilistic SA are discussed in page *DiscWhyProbabilisticSA*.

Probabilistic SA requires a probability distribution to be assigned to the simulator inputs. We first present some notation and will then discuss the interpretation of the probability distribution and the relevant measures of sensitivity in probabilistic SA.

Notation

In accordance with the standard *toolkit notation*, we denote the simulator by f and its inputs by x. The focus of SA is the relationship between x and the simulator output(s) $f(x)$. Since SA also typically tries to isolate the influences of individual inputs, or groups of inputs, on the output(s), we let x_j be the j-th element of x and will refer to this as the j-th input, for $j = 1, 2, \ldots, p$, where as usual p is the number of inputs. If J is a subset of the indices $\{1, 2, \ldots, p\}$, then x_J will denote the corresponding subset of inputs. For instance, if $J = \{2, 6\}$ then $x_J = x_{\{2,6\}}$ comprises inputs 2 and 6. Finally, x_{-j} will denote the whole of the inputs x *except* x_j, and similarly x_{-J} will be the set of all inputs except those in x_J.

We denote the probability distribution over the inputs x by ω. Formally, $\omega(x)$ is the joint probability density function for all the inputs. The marginal density function for input x_j is denoted by $\omega_j(x_j)$, while for the group of inputs x_J the density function is $\omega_J(x_J)$. The conditional distribution for x_{-J} given x_J is $\omega_{-J|J}(x_{-J} \,|\, x_J)$. Note, however, that it is common for the distribution ω to be such that the various inputs are statistically independent. In this case the conditional distribution $\omega_{-J|J}$ does not depend on x_J and is identical to the marginal distribution ω_{-J}.

Random X

Note that by assigning probability distributions to the inputs in probabilistic SA we formally treat those inputs as random variables. Notationally, it is conventional in statistics to denote random variables by capital letters, and this distinction is useful also in probabilistic SA. Thus, the symbol X denotes the set of inputs when regarded as random (i.e. uncertain), while x continues to denote a particular set of input values. Similarly, X_J represents those random inputs with subscripts in the set J, while x_J denotes an actual value for those inputs.

It is most natural to think of X as a random variable in the context of the third and fourth uses of SA, as listed above. When there is genuine uncertainty about the proper values to assign to inputs in order to obtain the output(s) of interest, then X can indeed be interpreted as random, and $\omega(x)$ is then the probability density function describing the relative probabilities of different possible values x for X. SA then involves trying to understand the role of uncertainty about the various inputs in the induced uncertainty concerning the outputs $f(X)$ or concerning a decision based on these outputs.

Interpretation of $\omega(x)$ as a weight function

However, in the context of other uses of SA it may be less natural to think of X as random. When our objective is to gain understanding of the simulator's behaviour or to identify inputs that are more or less redundant, it is not necessary to regard the inputs as uncertain. It is, nevertheless, important to think about the range of input values over which we wish to achieve the desired understanding or dimension reduction. In this case, $\omega(x)$ can simply define that range by being zero for any x outside the range. Within the range of interest, we may regard $\omega(x)$ as a weight function. Whilst we might normally give equal weight to all points in the range, for some purposes it may be appropriate to give more weight to some points than to others.

Whether we regard $\omega(x)$ as defining a probability distribution or simply as a weight function, it allows us to average over the region of interest to define measures of sensitivity.

Probabilistic SA methods

We have seen that different uses of SA may suggest not only different ways of interpreting the $\omega(x)$ function but also may demand different kinds of SA measures. However, there are similarities and connections between the various measures, particularly between measures used for understanding, dimension reduction and analysing output uncertainty. These are discussed together in page *DiscVarianceBasedSA* (with some technical details in page *DiscVarianceBasedSATheory*). Ways to use these variance based SA measures for output uncertainty are considered in page *DiscSensitivityAndOutputUncertainty*. Their usage for understanding and dimension reduction is discussed in page *DiscSensitivityAndSimplification*.

Measures specific to analysing decision uncertainty are presented in the discussion page *DiscDecisionBasedSA*, where the variance based measures are also shown to arise as a special case, while the discussion page *DiscSensitivityAndDecision* considers the practical use of these measures for decision uncertainty.

8.16.4 SA in the toolkit

All SA measures concern the relationship between a simulator's inputs and outputs. They generally depend on the whole function f and implicitly suppose that we know $f(x)$ for all x. In practice, we can only run the simulator at a limited number of input configurations, and as a result any computation of SA measures must be subject to computation error. The conventional Monte Carlo approach, for instance, involves randomly sampling input values and then running the simulator at each sampled x. Its accuracy can be quantified statistically and reduces as the sample size increases. For large and complex simulators, Monte Carlo may be infeasible because of the amount of computation required. One of the motivations for the MUCM approach is that tasks such as SA can be performed much more efficiently, first building an emulator using a modest number of simulator runs, and then computing the SA measures using the emulator. The computation error is then quantified in terms of *code uncertainty*.

SA is one of the tasks that we aim to cover in each of the main threads (i.e. core threads, variant threads and generic threads). Each of these threads describes the modelling and building of an emulator for a particular kind of simulator, and each explains how to use that emulator to carry out tasks associated with that simulator. So wherever the procedure for computing SA measures has been worked out for a particular thread, that procedure will be described in that thread. See the page *DiscToolkitSensitivityAnalysis* for a discussion of which procedures are available in which threads.

8.16.5 Additional comments

Note that although SA is usually presented as being concerned with the relationship between simulator inputs and outputs, the principal purpose of a simulator is to represent a particular real-world phenomenon: it is often built to explore how that real phenomenon behaves and some or all of its inputs represent quantities in the real world. The simulator output $f(x)$ is intended to predict the value of some aspect of the real phenomenon when the corresponding real quantities take values x. So in principle we may wish to consider SA in which the simulator is replaced by reality. This may be developed in a later version of this toolkit.

In some application areas, the term "probabilistic sensitivity analysis" is used for what we call uncertainty analysis.

8.16.6 References

Three books are extremely useful guides to the uses of SA in practice, and for non-MUCM methods for computing some of the most important measures.

Saltelli, A., Chan, K. and Scott, E. M. (eds.) (2000). Sensitivity Analysis. Wiley.

Saltelli, A., Tarantola, S., Campolongo, F. and Ratto, M. (2004). Sensitivity Analysis in Practice: A guide to assessing scientific models. Wiley.

Saltelli, A., Ratto, M., Andres, T., Campolongo, F., Cariboni, J., Gatelli, D., Saisana, M. and Tarantola, S. (2008). Global Sensitivity Analysis: The primer. Wiley.

MUCM methods for computing SA measures are based on using emulators. The basic theory was presented in

Oakley, J. E. and O'Hagan, A. (2004). Probabilistic sensitivity analysis of complex models: a Bayesian approach. *Journal of the Royal Statistical Society Series B* 66, 751-769. (Online)

Whilst the above references are all useful for background information, the toolkit pages present a methodology for efficient SA using emulators that is not published elsewhere in such a comprehensive way.

8.17 Thread: Dynamic Emulation

8.17.1 Overview

This thread describes how to construct an *emulator* for a *dynamic simulator*. Here, we describe an approach based on emulating what we call the *single step function*. An alternative, though potentially less practical approach is to build a multiple output emulator using the techniques described in the variant thread for analysing a simulator with multiple inputs (*ThreadVariantMultipleOutputs*). For a comparison of the two methods see the alternatives page on dynamic emulation approaches (*AltDynamicEmulationApproach*).

Readers should be familiar with the analysis of the core model, as described in the relevant core thread (*ThreadCoreGP*), before continuing here. The task of building the emulator of the single step function is an example of *the core problem* (though the multivariate extension described in *ThreadVariantMultipleOutputs* will be necessary in most cases). This thread does not currently describe the analysis for dynamic simulators within the *Bayes linear* framework, but methods will be added to these pages as they are developed.

8.17.2 Simulator specifications and notation

We make the distinction between the **full simulator** and the **simulator single step function**. The full simulator is the simulator that we wish to analyse, and has the following inputs and outputs.

Full simulator inputs:

- Initial conditions w_0.
- A time series of external *forcing inputs* $a_1, \dots, a_T$.
- Constant parameters ϕ.

Full simulator outputs:

- A time series $w_1, \dots, w_T$.

We refer to w_t as the *state variable* at time t. Each of w_t, a_t and ϕ may be scalar or vector-valued. The value of T may be fixed, or varied by the simulator user. We represent the full simulator by the function $(w_1, \dots, w_T) = f_{full}(w_0, a_1, \dots, a_T, \phi)$. We define $\mathcal{X}_{full}$ to be the input region of interest for the full simulator, with a point in $\mathcal{X}_{full}$ represented by $(w_0, a_1, \dots, a_T, \phi)$.

The full simulator produces the output $(w_1, \dots, w_T)$ by iteratively applying a function of the form $w_t = f(w_{t-1}, a_t, \phi)$, with $f(.)$ known as the single step function. Inputs and outputs for the single step functions are therefore as follows.

Single step function inputs:

- The current value of the state variable w_{t-1} .
- The associated forcing input a_t.
- Constant parameters ϕ.

Single step function output:

- The value of the state variable at the next time point w_t.

In this thread, we require the user to be able to run the single step function at any input value. Our method for emulating the full simulator is based on emulating the single step function. We define $\mathcal{X}_{single}$ to be the input region of interest for the single step function, with a point in $\mathcal{X}_{single}$ represented by $x = (w, a, \phi)$.

Training inputs for the single step emulator are represented by $D = \{x_1, \ldots, x_n\}$. Note that the assignment of indices $1, \ldots, n$ to the training inputs is arbitrary, so that there is no relationship between x_i and x_{i+1}.

8.17.3 Emulating the full simulator: outline of the method

1. Build the single step emulator: an emulator of the single step function.
2. Iterate the single step emulator to randomly sample $w_1, \ldots, w_T$ given a full simulator input $(w_0, a_1, \ldots, a_T, \phi)$. Repeat for different choices of full simulator input within $\mathcal{X}_{full}$.
3. Inspect the distribution of sampled trajectories $w_1, \ldots, w_T$ obtained in step 2 to determine whether the training data for the single step emulator are adequate. If necessary, obtain further runs of the single step function and return to step 1.

8.17.4 Step 1: Build an emulator of the single step function $w_t = f(w_{t-1}, a_t, \phi)$

This can be done following the procedures in *ThreadCoreGP*, (or *ThreadVariantMultipleOutputs* if the state variable is a vector). Two issues to consider in particular are the choice of mean function, and the design for the training data.

1) Choice of single step emulator mean function

(See the alternatives page on emulator prior mean function (*AltMeanFunction*) for a general discussion of the choice of mean function). The user should think carefully about the relationship between w_t and (w_{t-1}, a_t, ϕ). The state variable at time t is likely to be highly correlated with the state variable at time $t-1$, and so the constant mean function is unlikely to be suitable.

2) Choice of single step emulator *design*

Design points for the single step function can be chosen following the general principles in the alternatives page on training sample design for the core problem (*AltCoreDesign*). However, there is one feature of the dynamic emulation case that is important to note: we can get feedback from the emulator to tell us if we have specified the input region of interest $\mathcal{X}_{single}$ appropriately. If the emulator predicts that w_t will move outside the original design space for some value of t, then we will want to predict $f(w_t, a_{t+1}, \phi)$ for an input (w_t, a_{t+1}, ϕ) outside our chosen $\mathcal{X}_{single}$. Alternatively, we may find that the state variables are predicted to lie in a much smaller region than first thought, so that some training data points may be wasted. Hence it is best to choose design points sequentially; we choose a first set based on our initial choice of $\mathcal{X}_{single}$, and then in steps 2 and 3 we identify whether further training runs are necessary.

We have not yet established how many training runs are optimal at this stage (or the optimal proportion of total training runs to be chosen at this stage), though this will depend on how well $\mathcal{X}_{single}$ is chosen initially. In the application in Conti et al (2009), with three state variable and two forcing inputs, we found the choice of 30 initial training runs and 20 subsequent training runs to work well.

As we will need to iterate the single step emulator over many time steps, we emphasise the importance of *validating* the emulator, using the procedure page on validating a Gaussian process emulator (*ProcValidateCoreGP*).

8.17.5 Step 2: Iterate the single step emulator over the full simulator input region of interest

We now iterate the single step emulator to establish whether the initial choice of design points D is suitable . We do so by choosing points from $\mathcal{X}_{full}$, and iterating the single step emulator given the specified $(w_0, a_1, \ldots, a_T, \phi)$. A procedure for doing so is described in *ProcExploreFullSimulatorDesignRegion*.

8.17.6 Step 3: Inspect the samples from step 2 and choose additional training runs

Following step 2, we have now have samples $(w_{t-1}^{(i)}, a_t^{(i)}, \phi^{(i)})$ for $t = 1, \ldots, T$ and $i = 1, \ldots, N$. These samples give us a revised assessment of $\mathcal{X}_{single}$, as the simulation in step 2 has suggested that we wish to predict $f(.)$ at each point $(w_{t-1}^{(i)}, a_t^{(i)}, \phi^{(i)})$. We now compare this collection of points with the original training design D to see if additional training data are necessary. If further training data are obtained, we re-build the single step emulator and return to step 2.

We do not currently have a simple procedure for choosing additional training data, as the shape of $\mathcal{X}_{single}$ implied by the sampled $(w_{t-1}^{(i)}, a_t^{(i)}, \phi^{(i)})$ is likely to be quite complex. A first step is to compare the marginal distribution of each state vector element in the sample with the corresponding elements in the training design D, as this may reveal obvious inadequacies in the training data. It is also important to identify the time t^* when a sampled time series $(w_{t-1}^{(i)}, a_t^{(i)}, \phi^{(i)})$ for $t = 1, \ldots, T$ *first* moves outside the design region. The single step emulator may validate less well the further the input moves from the training data, so that samples $(w_{t-1}^{(i)}, a_t^{(i)}, \phi^{(i)})$ for $t > t^*$ may be less 'reliable'.

8.17.7 Tasks

Having obtained a satisfactorily working emulator, the MUCM methodology now enables efficient analysis of a number of tasks that regularly face users of simulators.

Prediction

The simplest of these tasks is to use the emulator as a fast surrogate for the simulator, i.e. to predict what output the simulator would produce if run at a new point in the input space. We have two methods for doing this: the exact simulation method described in the procedure page *ProcExactIterateSingleStepEmulator* (used in step 2 in the construction of the emulator) and an approximation described in the procedure page *ProcApproximateIterateSingleStepEmulator* which can be faster to implement. (See the alternatives page *AltIteratingSingleStepEmulators* for a comparison of the two).

Uncertainty analysis

Uncertainty analysis is the process of predicting the simulator output when one or more of the inputs are uncertain. The procedure page on uncertainty analysis for dynamic emulators (*ProcUADynamicEmulator*) explains how this is done.

8.17.8 Additional Comments, References, and Links

Methods for other tasks such as *sensitivity analysis* will be added to these pages as they are developed.

The methodology described here is based on

Conti, S., Gosling, J. P., Oakley, J. E. and O'Hagan, A. (2009). Gaussian process emulation of dynamic computer codes. *Biometrika* 96, 663-676.

8.18 Thread Variant: Linking Models to Reality using Model Discrepancy

8.18.1 Overview

The principal user entry points to the MUCM toolkit are the various threads, as explained in the Toolkit structure page (*MetaToolkitStructure*). This thread takes the user through possible techniques used to link the model to reality. This is a vital step in order to incorporate observed data and to make any kind of statement about reality itself. As the process of linking model to reality is somewhat independent of the *emulation* strategy used, we will assume that the user has successfully emulated the model using one of the strategies outlined in the main threads (see e.g. *ThreadCoreGP* or *ThreadCoreBL*). If the model is fast enough, emulation may not even be required, but the process described below should still be employed. Here we use the term model synonymously with the term *simulator*.

As we are not concerned with the details of emulation we will describe a substantially more general case than covered by the *core model*. Assumptions we share with the core model are:

- We are only concerned with one simulator.
- The simulator is deterministic.

However, in contrast to the Core model we now assume:

- We have observations of the real world process against which to compare the simulator.
- We wish to make statements about the real world process.
- The simulator can produce one, or more than one output of interest.

The first two assumptions are the main justifications for this thread. The third point states that we will be dealing with both univariate and multivariate cases. We will also discuss both the fully Bayesian case, where the simulator is represented as a Gaussian process (*ThreadCoreGP*), and the Bayes Linear case (*ThreadCoreBL*), where our beliefs about the simulator are represented as a *second-order specification*. Other assumptions such as whether simulator derivative information is available (which might have been used in the emulation construction process), do not concern us here.

8.18.2 Notation

In accordance with *toolkit notation*, in this page we use the following definitions:

- x - vector of inputs to the model
- $f(x)$ - vector of outputs of the model function
- y - vector of the actual system values
- z - vector of observations of reality y
- x^+ - the 'best input'
- d - the model discrepancy

8.18.3 Model Discrepancy

No matter how complex a particular model of a physical process is, there will always be a difference between the outputs of the model and the real process that the model is designed to represent. If we are interested in making statements about the real system using results from the model, we must incorporate this difference into our analysis. Failure to do so could lead to grossly inaccurate predictions and inferences regarding the structure of the real system. See the discussion page on model discrepancy (*DiscWhyModelDiscrepancy*) where these ideas are detailed further.

We assume that the simulator produces r outputs that we are interested in comparing to real observations. One of the simplest and most popular methods to represent the difference between model and reality is that of the *Best Input Approach* which defines the Model Discrepancy via:

$$y = f(x^+) + d,$$

where y, $f(x)$, d are all random r-vectors representing the system values, the simulator outputs and the *Model Discrepancy* respectively. x^+ is the vector of '*Best Inputs*', which represents the values that the input parameters take in the real system. We consider d to be independent of x^+ and uncorrelated with f and f^+ (in the Bayes Linear Case) or independent of f (in the fully Bayesian Case), where $f^+ = f(x^+)$. Note that the r-vector d may still posses a rich covariance structure, which will need to be *assessed*. Although the Best Input approach is often chosen for its simplicity, there are certain subtleties in the definition of x^+ and in the independence assumptions. A full discussion of this approach is given in the discussion page on the best input approach (*DiscBestInput*), and also see *DiscWhyModelDiscrepancy* for further general discussion on the need for a Model Discrepancy term.

More careful methods have been developed that go beyond the simple assumptions of the Best Input Approach. One such method, known as *Reification*, is described in the discussion page *DiscReification* with further theoretical details given in *DiscReificationTheory*.

8.18.4 Observation Equation

Unfortunately, we are never able to measure the real system values represented by the vector y. Instead, we can perform measurements z of y that involve some measurement error. A simple way to express the link between z and y is using the observation equation:

$$z = y + e$$

where we assume that the measurement error e is uncorrelated with y (in the fully Bayesian case). It maybe the case that z does not correspond exactly to y; for example, z could correspond to either a subset or some linear combination of the elements of the vector y. Methods for dealing with these cases where $z = Hy + e$, for some matrix H, and cases where z is a more complex function of y are described in the discussion page on the observation equation (*DiscObservations*).

8.18.5 Assessing the Model Discrepancy

In order to make statements about the real system y, we need to be able to *assess* the Model Discrepancy d. Assessing or estimating d is a difficult problem: as is discussed in *DiscWhyModelDiscrepancy* d represents a statistical model of a difference which is in reality very complex. Various strategies are available, the suitability of each depending on the context of the problem.

The first is that of Expert assessment, where the modeller's beliefs about the deficiencies of the model are converted into statistical statements about d (see *DiscExpertAssessMD*). Such considerations are always important, but they are of particular value when there is a relatively small amount of observational data to compare the model output to.

The second is the use of informal methods to obtain order of magnitude assessments of d (see *DiscInformalAssessMD*). These would often involve the use of simple computer model experiments to assess the contributions to the model discrepancy from particular sources (e.g. forcing function uncertainty).

The third is the use of more formal statistical techniques to assess d. These include Bayesian inference (for example, using MCMC), Bayes Linear inference methods and Likelihood inference. Although more difficult to implement, these methods have the benefit of rigour (see *DiscFormalAssessMD* for details). It is worth noting that a full Bayesian inference would *calibrate* the model and assess d simultaneously.

8.18.6 Cases Where Discrepancy has Clearly Defined Structure.

Physical Structure

The structure of the discrepancy vector corresponds to the underlying structure of the output vector, and we often choose to make aspects of this structure explicit in our notation. Often such structures are physical in nature, for example various parts of the system could be naturally labeled by their space-time location u. Then we might define the model discrepancy via:

$$y(u) = f(u, x^+) + d(u)$$

where u labels the space-time location of the system, model and model discrepancy. Note that there may still be multiple outputs at each value of u.

Consideration of such structures is important as they suggest natural ways of parameterising the covariance matrix of $d(u)$, for example using a *separable form*, and they can also suggest building certain physical trends into $\mathrm{E}[d]$. Further discussion and examples of structured model discrepancies can be found in *DiscStructuredMD*.

Exchangeable Models

In some situations a simulator may require, in addition to the usual input parameters x, a specification of certain system conditions. The most common example of a system condition is that of a forcing function (e.g. rainfall in a flood model). Often there exists a set of different system conditions (e.g. a set of different possible realisations of rainfall over a fixed time period) that are considered equivalent in some sense. It can then be appropriate to consider the simulator, run at each of the choices of system condition, as a set of Exchangeable Computer Models. In this case the structure of the model discrepancy has a particular form, and methodology has been developed to analyse this more complex situation and the subsequent link to reality, as can be found in the discussion page on exchangeable models (*DiscExchangeableModels*).

8.18.7 Additional Comments, References and Links.

This thread has described the importance of including model discrepancy, and discussed methods of assessing such a term. In the next release, several procedures will be described for which model discrepancy plays a vital role. These will include Calibration, History Matching and Prediction.

8.19 Thread: Analysis of a simulator with multiple outputs using Gaussian Process methods

8.19.1 Overview

The multivariate emulator

The principal user entry points to the *MUCM* toolkit are the various *threads*, as explained in the Toolkit structure page (*MetaToolkitStructure*). The main threads give detailed instructions for building and using *emulators* in various contexts.

This thread takes the user through the analysis of a variant of the most basic kind of problem, using the fully Bayesian approach based on a Gaussian process (GP) emulator. We characterise the basic multi-output model as follows:

- We are only concerned with one *simulator*.
- The output is *deterministic*.

- We do not have observations of the real world process against which to compare the simulator.
- We do not wish to make statements about the real world process.
- We cannot directly observe derivatives of the simulator.

Each of these requirements is also a part of the core problem, and is discussed further in *DiscCore*. However, the core problem further assumes that the simulator only produces one output, or that we are only interested in one output. We relax that assumption here. The core thread *ThreadCoreGP* deals with the analysis of the core problem using a GP emulator. This variant thread extends the core analysis to the case of a simulator with more than one output.

The fully Bayesian approach has a further restriction:

- We are prepared to represent the simulator as a *Gaussian process*.

There is discussion of this requirement in *DiscGaussianAssumption*.

Alternative approaches to emulation

There are various approaches to tackling the problems raised by having multiple outputs, which are discussed in the alternatives page on emulating multiple outputs (*AltMultipleOutputsApproach*). Some approaches reduce or transform the multi-output model so that it can be analysed by the methods in *ThreadCoreGP*. However, others employ a *multivariate GP* emulator that is described in detail in the remainder of this thread.

8.19.2 The GP model

The first stage in building the emulator is to model the mean and covariance structures of the Gaussian process that is to represent the simulator. As explained in the definition of a *multivariate Gaussian process*, a GP is characterised by a mean function and a covariance function. We model these functions to represent *prior* beliefs that we have about the simulator, i.e. beliefs about the simulator prior to incorporating information from the *training sample*.

Alternative choices of the emulator prior mean function are considered in *AltMeanFunction*, with specific discussion on the multivariate case in *AltMeanFunctionMultivariate*. In general, the choice will lead to the mean function depending on a set of *hyperparameters* that we will denote by β. We will generally write the mean function as $m(\cdot)$ where the dependence on β is implicit. Note that if we have r outputs, then $m(\cdot)$ is a vector of $1 \times r$ elements comprising the mean functions of the various outputs.

The most common approach is to define the mean function to have the linear form $m(x) = h^T(x)\beta$, where $h(\cdot)$ is a $q \times 1$ vector of regressor (or basis) functions whose specification is part of the choice to be made. Note that β is a $q \times r$ matrix.

The covariance function for a multivariate GP specifies the $r \times r$ covariance matrix between the r outputs of the simulator at an input configuration x and the r outputs at input x'. A number of options of varying complexity are available for the covariance function, which are discussed in *AltMultivariateCovarianceStructures*. The hyperparameters in a general covariance function, including the hyperparameters in the correlation function and scale parameters in the covariance function are denoted by ω.

The techniques that follow in this thread will be expressed as far as possible in terms of the general forms of the mean and covariance functions, depending on general hyperparameters β and ω. However, in many cases, simpler formulae and methods can be developed when the linear mean function and the separable covariance function with a Gaussian form of correlation function are are chosen, and some techniques in this thread may only be available in the special cases.

8.19.3 Prior distributions

The GP modelling stage will have described the mean and covariance structures in terms of some hyperparameters. A fully Bayesian approach now requires that we express probability distributions for these that are again *prior* distributions. Alternative forms of prior distributions for GP hyperparameters are discussed in *AltGPPriors*, with some specific suggestions for the covariance function hyperparameters ω given in *AltMultivariateCovarianceStructures*. The result is in general a joint (prior) distribution $\pi(\beta, \omega)$. Where required, we will denote the marginal distribution of ω by $\pi_\omega(\cdot)$, and similarly for marginal distributions of other groups of hyperparameters. This alternatives for multivariate GP priors for the input-output *separable* case are further discussed in *AltMultivariateGPPriors*.

8.19.4 Design

The next step is to create a *design*, which consists of a set of points in the input space at which the simulator is to be run to create the training sample. Design options for the core problem are discussed in *AltCoreDesign*. Design for the multiple output problem has not been explicitly studied, but we believe that designs for the core problem will be good also for the multi-output problem, although it seems likely that a larger number of design points could be required.

The result of applying one of the design procedures described there is an ordered set of points $D = \{x_1, x_2, \ldots, x_n\}$. The simulator is then run at each of these input configurations, producing a $n \times r$ matrix of outputs. The i-th column of this matrix is the output produced by the simulator from the run with inputs x_i.

One suggestion that is commonly made for the choice of the sample size n is $n = 10p$, where p is the number of inputs. (This may typically be enough to obtain an initial fit, but additional simulator runs are likely to be needed for the purposes of *validation*, and then to address problems raised in the validation diagnostics as discussed below. In general there are no sure rules of thumb for this choice and careful validation is critical in building an emulator.)

8.19.5 Fitting the emulator

Given the training sample and the GP prior model, the process of building the emulator is theoretically straightforward, and is set out in the procedure page for building a multivariate Gaussian process emulator for the core problem (*ProcBuildMultiOutputGP*). If a separable covariance function is chosen, then there are various simplifications to the procedure, set out in *ProcBuildMultiOutputGPSep*.

The result of *ProcBuildMultiOutputGP* is the emulator, fitted to the prior information and training data. It is in the form of an updated multivariate GP (or, in the separable covariance case, a related process called a *multivariate t-process*) conditional on hyperparameters, plus one or more sets of representative values of those hyperparameters. Addressing the tasks below will then consist of computing solutions for each set of hyperparameter values (using the multivariate GP or t-process) and then an appropriate form of averaging of the resulting solutions (see the procedure page on predicting simulator outputs using a GP emulator (*ProcPredictGP*)).

Although the fitted emulator will correctly represent the information in the training data, it is always important to validate it against additional simulator runs. The procedure of validating a Gaussian process emulator is described in *ProcValidateCoreGP*. It is often necessary, in response to the validation diagnostics, to rebuild the emulator using additional training runs.

Validating multi-output emulators is more challenging. The most simple approach is to validate on individual outputs (ignoring any correlations between them implied by the covariance function) using the methods defined in *ProcValidateCoreGP*. This is not the full answer, however there is relatively little experience of validating multivariate emulators in the literature. We hope to develop this and include insights in future releases of the toolkit.

8.19.6 Tasks

Having obtained a working emulator, the MUCM methodology now enables efficient analysis of a number of tasks that regularly face users of simulators.

Prediction

The simplest of these tasks is to use the emulator as a fast surrogate for the simulator, i.e. to predict what output the simulator would produce if run at a new point x' in the input space. The process of predicting one or more new points is set out in *ProcPredictGP*.

For some of the tasks considered below, we require to predict the output not at a set of discrete points, but in effect the entire output function as the inputs vary over some range. This can be achieved also using simulation, as discussed in the procedure page for simulating realisations of an emulator (*ProcSimulationBasedInference*).

Sometimes interest will be in a deterministic function of one or more of the outputs. If your only interest is in a function of a set of outputs which is a pre-determined mapping, building a direct single output emulator is probably the most efficient approach. In other situations, such as when you are interested in both the raw outputs and one or more functions of the outputs, or when you are interested in function(s) that depend some auxiliary variables other than just the raw outputs of the simulator, then it is better to build the multivariate emulator first, then use the procedure for obtaining a sample from the predictive distribution of the function set out in *ProcPredictMultiOutputFunction*.

It is worth noting that the predictive distribution (i.e. emulator) of any **linear** transformation of the outputs is also a multivariate GP, or t-process. In particular, the emulator of a single linear combination of outputs is a regular univariate GP emulator, so all the core theory applies whenever we want to do anything with a single output or with a single linear combination of outputs. It is important to realise that having built a multivariate emulator, these single linear output functions are derived from it, not by fitting a univariate emulator separately (which will almost certainly produce slightly different results).

Uncertainty analysis

Uncertainty analysis is the process of predicting the simulator output when one or more of the inputs are uncertain.

Sensitivity analysis

In *sensitivity analysis* the objective is to understand how the output responds to changes in individual inputs or groups of inputs. The most common approach is a *variance based* sensitivity analysis.

8.19.7 Examples

ExamMultipleOutputs is an example demonstrating the multivariate emulator with a number of different covariance functions. *ExamMultipleOutputsPCA* is a more complex example showing a reduced dimension multivariate emulator applied to a chemometrics model using PCA to reduce the dimension of the output space.

8.19.8 Additional Comments, References, and Links

Other tasks that can be addressed include optimisation (finding the values of one or more inputs that will minimise or maximise the output) and decision analysis (finding an optimal decision according to a formal description of utilities). A related task is *decision-based* sensitivity analysis. We expect to add procedures for these tasks for the core problem in due course.

Another task that is very often required is *calibration*. This requires us to think about the relationship between the simulator and reality, which is dealt with in *ThreadVariantModelDiscrepancy*. Although the calibration task itself is not covered in this release of the Toolkit we hope to include it in a future release.

8.20 Thread Variant: Two-level emulation of the core model using a fast approximation

8.20.1 Overview

Often, a *computer model* can be evaluated at different levels of accuracy resulting in different versions of the computer model of the same system. For example, this can arise from simplifying the underlying mathematics, by adjusting the model gridding, or by changing the accuracy of the model's numerical solver. Lower accuracy models can often be evaluated for a fraction of the cost of the full computer model, and may share many qualitative features with the original. Often, the coarsened computer model is informative for the accurate computer model, and hence for the physical system itself. By using evaluations of the coarse model in addition to those of the full model, we can construct a single multiscale *emulator* of the computer simulation.

When an approximate version of the simulator is available, we refer to it as the *coarse simulator*, $f^c(\cdot)$, and the original as the *accurate simulator*, $f^a(\cdot)$, to reflect these differences in precision. We consider evaluations of the coarse model to be relatively inexpensive when compared to $f^a(x)$. In such a setting, we can obtain many evaluations of the coarse simulator and use these to construct an informed emulator for the coarse model. This provides a basis for constructing an informed prior specification for the accurate emulator. We then select and evaluate a small number of accurate model runs to update our emulator for the accurate model. This transfer of beliefs from coarse to accurate emulator is the basis of *multilevel emulation* and is the focus of this thread.

This thread considers the problem of constructing an emulator for a single-output deterministic computer model $f^a(x)$ when we have access to a single approximate version of the same simulator, $f^c(x)$.

8.20.2 Requirements

The requirements for the methods and techniques described in this page differ from the *core problem* by relaxing the requirement that we are only concerned with a single simulator. We now generalise this problem to the case where

- We have two computer models for the same complex system - one of the models (the *coarse simulator*) is comparatively less expensive to evaluate, though consequently less accurate, than the second simulator (the *accurate simulator*)

8.20.3 General process

The general process of two-level emulation follows three distinct stages:

1. **Emulate the coarse model** - *design* for and perform *evaluations* of $f^c(x)$, use these evaluations to construct an emulator for the coarse model
2. **Link the coarse and accurate models** - by parametrising the relationship between the two simulators, we use information from the coarse emulator to build a prior emulator for $f^a(x)$
3. **Emulate the accurate model** - design for and evaluate a small number of runs of $f^a(x)$, use these evaluations to update the prior emulator

Stages 1 and 3 in this process are applications of standard emulation methodology discussed extensively in the Toolkit. Stage 2 is unique to multiscale emulation and the combination of information obtained from the emulator $f^c(x)$ with beliefs about the relationship between the two models is at the heart of this approach.

8.20.4 Emulating the coarse model

To represent our uncertainty about the high-dimensional coarse computer model $f^c(x)$, we build a coarse emulator for the coarse simulator. This gives an emulator of the form:

$$f^c(x) = m^c(x) + w^c(x)$$

where $m^c(x)$ represents the emulator *mean function*, and $w^c(x)$ is a stochastic residual process with a specified *covariance function*.

The choice of the methods of emulator construction depends on the problem. In the case where the coarse simulator is very fast and very large amounts of model evaluations can be obtained for little expense, we may consider a purely empirical method of emulator construction as described in the procedure for the empirical construction of a Bayes linear emulator (*ProcBuildCoreBLEmpirical*). When the coarse simulator requires a moderate amount of resource to evaluate (albeit far less than $f^a(x)$) and when appropriate prior beliefs about the model are available, then we can apply the *fully-Bayesian* methods of the thread for the analysis of the core model using Gaussian process methods (*ThreadCoreGP*) or the *Bayes linear* methods of *ThreadCoreBL*.

The manner in which we construct the emulator is not important, merely that we obtain an emulator as described in the form of either numerical estimates, *adjusted beliefs*, or posterior distributions for the emulator mean function parameters β^c, the variance/correlation hyperparameters $\{(\sigma^c)^2, \delta^c\}$, and an updated residual process. If the coarse emulator is built using Bayes linear methods, the necessary mean, variance and covariance specifications are provided within the relevant thread. Details of how to obtain the corresponding quantities for a fully-Bayesian emulator will be provided in a later release of the toollkit.

8.20.5 Linking the coarse and accurate emulators

Given that the coarse simulator is informative for the accurate simulator, we can use our coarse emulator as a basis for constructing our prior beliefs for the emulator of $f^a(x)$. To construct such a prior, we model the relationship between the two simulators and then combine information from $f^c(x)$ with appropriate belief specifications about this relationship. We express our emulator for the accurate model in a similar form as the coarse emulator

$$f^a(x) = m^a(x) + w^a(x),$$

In general, we express the accurate emulator in terms of either the coarse simulator itself or elements of the coarse simulator in conjunction with some additional parameters which capture how we believe the two simulators are related.

There are many ways to parametrise the relationship between the two computer models. Common approaches include:

Single multiplier: A simple approach to linking the computer models is to consider the accurate simulator to be a re-scaled version of the coarse simulator plus additional residual variation. This yields an accurate emulator of the form:

$$f^a(x) = \rho f^c(x) + w^{a'}(x),$$

where ρ is an unknown scaling parameter, and $w^{a'}(x)$ is a new stochastic residual process unique to the accurate computer model. We may consider the single multiplier method when we believe that the difference in behaviour between the two models is mainly a matter of scale, rather than changes in the shape or location of the output.

In this case, we can consider the mean function of the accurate emulator to be $m^a(x) = \rho m^c(x)$, and the residual process can be expressed as $w^a(x) = \rho w^c(x) + w^{a'}(x)$.

Regression multipliers: When the coarse emulator mean function takes a *linear form*, $m^c(x) = \sum_j \beta_j^c(x) h_j(x)$, the single multiplier method can be generalised. Instead of re-scaling the value of the coarse simulator itself, we can consider re-scaling the contributions from each of the regression *basis functions* to the emulator's mean function. This gives an accurate emulator of identical structure to the coarse emulator though with modified values of the regression coefficients,

$$f^a(x) = \sum_j \rho_j \beta_j^c h_j(x) + \rho_w w^c(x) + w^{a\prime}(x)$$

where ρ_j is an unknown scaling parameter for basis function $h_j(x)$, and ρ_w scales the contribution of the coarse residual process to the accurate emulator. We might choose to use this regression form, for example, when we consider that each term in the regression represents a physical process and the effects represented by $h_j(x)$ change as we move between the the two simulators.

In this case, we can consider the mean function of the accurate emulator to be $m^a(x) = \sum_j \beta_j^a h_j(x)$ where $\beta_j^a = \rho_j \beta_j^c$, and the residual process can be expressed as $w^a(x) = \rho_w w^c(x) + w^{a\prime}(x)$. In some cases it can be appropriate to express this relationship in the alternative form $\beta_j^a = \rho_j \beta_j^c + \gamma_j$, where γ_j is an additional unknown parameter. This alternative form can better accommodate models which have mean function effects which "switch on" as we move onto the accurate model.

When the mean function of the emulators has a linear form, the single multiplier method is a special case of the regression multipliers method obtained by setting $\rho_i = \rho^w = \rho$.

Spatial multiplier: Similar to the single multiplier method, we still consider the accurate simulator to be a re-scaling of the coarse simulator. However, the scaling factor is no longer a single unknown value but a stochastic process, $\rho(x)$, over the input space.

$$f^a(x) = \rho(x) f^c(x) + w^a(x).$$

This spatial multiplier approach is applicable when we expect the nature of the relationship between the two models to change as we move throughout the input space. Similarly to (1), we can write the mean function of the accurate emulator to be $m^a(x) = \rho(x) m^c(x)$, and the residual process can be expressed as $w^a(x) = \rho(x) w^c(x) + w^{a\prime}(x)$.

In general, we obtain a form for the accurate emulator given by the appropriate expressions for $m^a(x)$ and $w^a(x)$. Each of these components is expressed in terms of (elements of) the emulator for $f^c(x)$ and an additional residual process $w^a(x)$, and is parametrised by a collection of unknown linkage hyperparameters ρ.

8.20.6 Specifying beliefs about ρ_j and $w^a(x)$

Given the coarse emulator $f^c(x)$ and a model linking $f^c(x)$ to $f^a(x)$, then a prior specification for ρ and $w^a(x)$ are sufficient to develop a prior for the emulator for $f^a(x)$. In general, our uncertainty judgements about ρ and $w^a(x)$ will be problem-specific. For now, we describe a simple structure that these beliefs may take and offer general advice for making such statements.

We begin by considering that ρ_j and $w^{a\prime}(x)$ are independent of β^c and $w^c(x)$. The simplest general specification of prior beliefs for the multipliers ρ corresponds to considering that there exists no known systematic biases between the two models. This equates to the belief that the expected value of $m^a(x)$ is the same as $m^c(x)$, which implies

$$\mathrm{E}[\rho_j] = 1$$

The simplest specification for the variance and covariance of the ρ_j is to parametrise the variance matrix by two constants σ_ρ^2 and α such that

$$\mathrm{Var}[\rho_j] = \sigma_\rho^2$$
$$\mathrm{Corr}[\rho_j, \rho_k] = \alpha, i \neq j$$

where $\sigma_\rho^2 \geq 0$ and $\alpha \in [-1, 1]$. This belief specification is relatively simple. However by adjusting the value of σ_ρ^2 we can tighten or relax the strength of the relationship between the two simulators. By varying the value of α we can move from beliefs that the accurate simulator is a direct re-scaling of the coarse simulator (method (1) above) when $\alpha = 1$, to a model where the contribution from each of the regression basis functions varies independently when $\alpha = 0$. Specification of these values will typically come from expert judgement. However, performing a small number of paired evaluations on the two simulators and assessing the degree of association can prove informative when specifying values of σ_ρ^2 and α. Additionally, considering heuristic statements can be insightful - for example, the belief that it is highly unlikely that β_j^a has a different sign to β_j^c might suggest the belief that $3\text{sd}[\rho_j] = 1$.

For method (3), the corresponding beliefs would be that the prior mean of the stochastic process $\rho(x)$ was the constant 1, and that the prior variance was σ_ρ^2 with a given correlation function (likely of the same form as $w^c(x)$).

Beliefs about $w^{a\prime}(x)$ are more challenging to structure. In general, we often consider that the $w^{a\prime}(x)$ behaves similarly to $w^c(x)$ and so has a zero mean, variance $(\sigma^a)^2$, and the same correlation function and hyperparameter values as $w^c(x)$. More complex belief specifications can be used when we have appropriate prior information relevant to those judgements. For example, we may wish to have higher values of $\text{Corr}[\rho_j, \rho_k]$ when $h_j(x)$ and $h_k(x)$ are functions of the same input parameter or are of similar functional forms.

8.20.7 Constructing the prior emulator for $f^a(x)$

In the *Bayes linear* approach to emulation, our prior beliefs about the emulator $f^a(x)$ are defined entirely by the *expectation and variance*. Using the regression multiplier method (2) of linking the simulators, these beliefs are as follows:

$$\text{E}[f^a(x)] = \sum_j \text{E}[\rho_j\beta_j^c]h_j(x) + \text{E}[\rho_w w^c(x)] + \text{E}[w^{a\prime}(x)]$$

$$\text{Var}[f^a(x)] = \sum_j\sum_k h_j(x)h_k(x)\text{Cov}[\rho_j\beta_j^c, \rho_k\beta_k^c] + \text{Var}[\rho_w w^c(x)] + \text{Var}[w^{a\prime}(x)] + 2\sum_j h_j(x)\text{Cov}[\rho_j\beta_j^c, \rho_w w^c(x)]$$

where the constituent elements are either expressed directly in terms of our beliefs about ρ_j and $w^{a\prime}(x)$, or are obtained from the expressions below:

$$\begin{aligned}
\text{E}[\rho_j\beta_j^c] &= \text{E}[\rho_j]\text{E}[\beta_j^c] \\
\text{E}[\rho_w w^c(x)] &= \text{E}[\rho_w]\text{E}[w^c(x)] \\
\text{Var}[\rho_w w^c(x)] &= \text{Var}[\rho_w]\text{Var}[w^c(x)] + \text{Var}[\rho_w]\text{E}[w^c(x)]^2 + \text{E}[\rho_w]^2\text{Var}[w^c(x)] \\
\text{Cov}[\rho_j\beta_j^c, \rho_k\beta_k^c] &= \text{Cov}[\rho_j, \rho_k]\text{Cov}[\beta_j^c, \beta_k^c] + \text{Cov}[\rho_j, \rho_k]\text{E}[\beta_j^c]\text{E}[\beta_k^c] + \text{Cov}[\beta_j^c, \beta_k^c]\text{E}[\rho_j]\text{E}[\rho_k] \\
\text{Cov}[\rho_j\beta_j^c, \rho_w w^c(x)] &= \text{Cov}[\rho_j, \rho_w]\text{Cov}[\beta_j^c, w^c(x)] + \text{Cov}[\rho_j, \rho_w]\text{E}[\beta_j^c]\text{E}[w^c(x)] + \text{Cov}[\beta_j^c, w^c(x)]\text{E}[\rho_j]\text{E}[\rho_w]
\end{aligned}$$

Expressions for the single multiplier approach are obtained by replacing all occurrences of ρ_j and ρ_w with the single parameter ρ and beliefs about that parameter are substituted into the above expressions with $\text{Corr}[\rho, \rho] = 1$. Similarly for the spatial multiplier method (3), ρ_j and ρ_w are replaced by the process $\rho(x)$.

In the case where the coarse simulator is well-understood, much of the uncertainties surrounding the coarse coefficients β_j^c and the coarse residuals $w^c(x)$ will be eliminated. Any unresolved variation on these quantities is often negligible in comparison to the other uncertainties associated with $f^a(x)$. In such cases, we may make the assumption that the β_j^c (and hence the $w^c(x)$) are known and thus substantially simplify the expressions for $\text{E}[f^a(x)]$ and $\text{Var}[f^a(x)]$ as follows:

$$\text{E}[f^a(x)] = \sum_j \text{E}[\rho_j]g_j(x) + \text{E}[\rho_w]w^c(x) + \text{E}[w^{a\prime}(x)]$$

$$\text{Var}[f^a(x)] = \sum_j\sum_k g_j(x)g_k(x)\text{Cov}[\rho_j, \rho_k] + w^c(x)^2\text{Var}[\rho_w] + \text{Var}[w^{a\prime}(x)] + 2\sum_j g_j(x)w^c(x)\text{Cov}[\rho_j, \rho_w]$$

where we define $g_j(x) = \beta_j^c h_j(x)$. These simplifications substantially reduce the complexity of the emulation calculations as now the only uncertain quantities in $f^a(x)$ are ρ_j, ρ_w and $w^{a\prime}(x)$.

These quantities are sufficient to describe the Bayes linear emulator of $f^a(x)$. The *Gaussian process* approach requires a probability distribution for $f^a(x)$, which will be taken to be Gaussian with the above specified mean and variance.

8.20.8 Design for the accurate simulator

We are now ready to make a small number of evaluations of the accurate computer model and update our emulator for the $f^a(x)$. Since the accurate computer model is comparatively very expensive to evaluate, this design will be small – typically far fewer runs than those available for the coarse simulator.

Due to the small number of design points, the choice of design is particularly important. If the cost of evaluating $f^a(x)$ permits, then a space-filling design for the accurate simulator would still be effective. However as the number of evaluations will be typically limited, we may consider seeking an optimal design which has the greatest effect in reducing uncertainty about $f^a(x)$. The general procedure for generating such a design is described in *ProcOptimalLHC*, where the design criterion is given by the adjusted (or posterior) variance of the accurate emulator given the simulator evaluations (see the procedure for building a Bayes linear emulator for the core problem (*ProcBuildCoreBL*) and the expression given above).

8.20.9 Building the accurate emulator

Our prior beliefs about the accurate emulator and the design and evaluations of the accurate simulator provide sufficient information to directly apply the Bayesian emulation methods described in *ThreadCoreBL* or *ThreadCoreGP*. Once we have constructed the accurate emulator we can then perform appropriate diagnostics and validation, and use the emulator as detailed for suitable post-emulation tasks.

8.21 Thread: Emulators with derivative information

8.21.1 Overview

This thread describes how we can use derivative information in addition to standard function output to build an *emulator*. As in the core thread for analysing the core problem using GP methods (*ThreadCoreGP*) the following apply:

- We are only concerned with one *simulator*.
- The simulator only produces one output, or (more realistically) we are only interested in one output.
- The output is *deterministic*.
- We do not have observations of the real world process against which to compare the simulator.
- We do not wish to make statements about the real world process.

Each of these aspects of the core problem is discussed further in page *DiscCore*.

However now, we also assume we can directly observe first derivatives of the simulator either through an *adjoint* model or some other technique. This thread describes the use of derivative information when building an emulator for the fully Bayesian approach only, thus we require the further restriction:

- We are prepared to represent the simulator as a *Gaussian process*.

There is discussion of this requirement in page *DiscGaussianAssumption*. If we want to adopt the *Bayes linear* approach it is still possible to include derivative information and this may be covered in a future release of the toolkit. Further information on this can be found in Killeya, M.R.H., (2004) “Thinking Inside The Box” Using Derivatives to Improve Bayesian Black Box Emulation of Computer Simulators with Applications to Compartmental Models. Ph.D. thesis, Department of Mathematical Sciences, University of Durham.

Readers should be familiar with *ThreadCoreGP*, before considering including derivative information.

8.21.2 Active inputs

As in *ThreadCoreGP*

8.21.3 The GP model

As in *ThreadCoreGP* the first stage in building the emulator is to model the mean and covariance structures of the Gaussian process that is to represent the simulator. As explained in the definition page of a Gaussian process (*DefGP*), a GP is characterised by a mean function and a covariance function. We model these functions to represent prior beliefs that we have about the simulator, i.e. beliefs about the simulator prior to incorporating information from the *training sample*. The derivatives of a Gaussian process remain a Gaussian process and so we can use a similar approach to *ThreadCoreGP* here.

The choice of the emulator prior mean function is considered in the alternatives page *AltMeanFunction*. However here we must ensure that the chosen function is differentiable. In general, the choice will lead to the mean function depending on a set of *hyperparameters* that we will denote by β.

The most common approach is to define the mean function to have the linear form $m(x) = h(x)^{\rm T}\beta$, where $h(\cdot)$ is a vector of regressor functions, whose specification is part of the choice to be made. As we are including derivative information in the training sample we must ensure that $h(\cdot)$ is differentiable. This will then lead to the derivative of the mean function: $\frac{\partial}{\partial x}m(x) = \frac{\partial}{\partial x}h(x)^{\rm T}\beta$. For appropriate ways to model the mean, both generally and in linear form, see *AltMeanFunction*.

The covariance function is considered in the discussion page *DiscCovarianceFunction* and here must be twice differentiable. Within the toolkit we will assume that the covariance function takes the form $\sigma^2 c(\cdot,\cdot)$, where σ^2 is an unknown scale hyperparameter and $c(\cdot,\cdot)$ is called the correlation function indexed by a set of correlation hyperparameters δ. The correlation then between a point, x_i, and a derivative w.r.t input k at x_j, (denoted by $x_j^{(k)}$), is $\frac{\partial}{\partial x_j^{(k)}}c(x_i,x_j)$. The correlation between a derivative w.r.t input k at x_i, (denoted by $x_i^{(k)}$), and a derivative w.r.t input l at x_j, (denoted by $x_j^{(l)}$), is $\frac{\partial^2}{\partial x_i^{(k)}\partial x_j^{(l)}}c(x_i,x_j)$. The choice of correlation function is considered in the alternatives page *AltCorrelationFunction*.

The most common approach is to define the correlation function to have the Gaussian form $c(x_i,x_j) = \exp\{-(x_i - x_j)^{\rm T}C(x_i-x_j)\}$, where C is a diagonal matrix with elements the inverse squares of the elements of the δ vector. The correlation then between a point, x_i, and a derivative w.r.t input k at point j, $x_j^{(k)}$, is:

$$\frac{\partial}{\partial x_j^{(k)}}c(x_i,x_j) = \frac{2}{\delta^2\{k\}}\left(x_i^{(k)} - x_j^{(k)}\right)\exp\{-(x_i-x_j)^{\rm T}C(x_i-x_j)\},$$

the correlation between two derivatives w.r.t input k but at points i and j is:

$$\frac{\partial^2}{\partial x_i^{(k)}\partial x_j^{(k)}}c(x_i,x_j) = \left(\frac{2}{\delta^2\{k\}} - \frac{4\left(x_i^{(k)} - x_j^{(k)}\right)^2}{\delta^4\{k\}}\right)\exp\{-(x_i-x_j)^{\rm T}C(x_i-x_j)\},$$

and finally the correlation between two derivatives w.r.t inputs k and l, where $k \neq l$, at points i and j is:

$$\frac{\partial^2}{\partial x_i^{(k)}\partial x_j^{(l)}}c(x_i,x_j) = \frac{4}{\delta^2\{k\}\delta^2\{l\}}\left(x_j^{(k)} - x_i^{(k)}\right)\left(x_i^{(l)} - x_j^{(l)}\right)\exp\{-(x_i-x_j)^{\rm T}C(x_i-x_j)\}$$

8.21.4 Prior distributions

As in *ThreadCoreGP*

8.21.5 Design

The next step is to create a *design*, which consists of a set of points in the input space at which the simulator or adjoint is to be run to create the training sample. Design options for the core problem are discussed in the alternatives page on training sample design (*AltCoreDesign*). Here though, we also need to decide at which of these points we want to obtain function output and at which points we want to obtain partial derivatives. This adds a further consideration when choosing a design option but as yet we don't have any specific design procedures which take into account the inclusion of derivative information.

If one of the design procedures described in *AltCoreDesign* is applied, the result is an ordered set of points $D = \{x_1, x_2, \ldots, x_n\}$. Given D, we would now need to choose at which of these points we want to obtain function output and at which we want to obtain partial derivatives. This information is added to D resulting in the design, $\tilde{D}$ of length $\tilde{n}$. A point in $\tilde{D}$ has the form (x, d), where d denotes whether a derivative or the function output is to be included at that point. The simulator, $f(\cdot)$, or the adjoint of the simulator, $\tilde{f}(\cdot)$, (depending on the value of each d), is then run at each of the input configurations.

One suggestion that is commonly made for the choice of the sample size, n, for the core problem is $n = 10p$, where p is the number of inputs. (This may typically be enough to obtain an initial fit, but additional simulator runs are likely to be needed for the purposes of *validation*, and then to address problems raised in the validation diagnostics.) There is not, however, such a guide for what $\tilde{n}$ might be. If we choose to obtain function output and the first derivatives w.r.t to all inputs at every location in the design, then we would expect that fewer than $10p$ locations would be required; how many fewer though, is difficult to estimate.

8.21.6 Fitting the emulator

Given the training sample of function output and derivatives, and the GP prior model, the process of building the emulator is given in the procedure page *ProcBuildWithDerivsGP*

The result of *ProcBuildWithDerivsGP* is the emulator, fitted to the prior information and training data. As with the core problem, the emulator has two parts, an updated GP (or a related process called a *t-process*) conditional on hyperparameters, plus one or more sets of representative values of those hyperparameters. Addressing the tasks below will then consist of computing solutions for each set of hyperparameter values (using the GP or t-process) and then an appropriate form of averaging of the resulting solutions.

Although the fitted emulator will correctly represent the information in the training data, it is always important to validate it against additional simulator runs. For the *core problem*, the process of validation is described in the procedure page *ProcValidateCoreGP*. Here, we are interested in predicting function output, therefore as in *ProcValidateCoreGP* we will have a validation design D' which only consists of points for function output; no derivatives are required and as such the simulator, $f(\cdot)$, not the adjoint, $\tilde{f}(\cdot)$, is run at each x'_j in D'. Then in the case of a linear mean function, weak prior information on hyperparameters β and σ, and a single posterior estimate of δ, the predictive mean vector, m^*, and the predictive covariance matrix, V^*, required in *ProcValidateCoreGP*, are given by the functions $m^*(\cdot)$ and $v^*(\cdot, \cdot)$ which are given in *ProcBuildWithDerivsGP*. We can therefore validate an emulator built with derivatives using the same procedure as that which we apply to validate an emulator of the core problem. It is often necessary, in response to the validation diagnostics, to rebuild the emulator using additional training runs which can of course, include derivatives. We hope to extend the validation process using derivatives as we gain more experience in validation diagnostics and emulating with derivative information.

8.21.7 Tasks

Having obtained a working emulator, the MUCM methodology now enables efficient analysis of a number of tasks that regularly face users of simulators.

Prediction

The simplest of these tasks is to use the emulator as a fast surrogate for the simulator, i.e. to predict what output the simulator would produce if run at a new point in the input space. In this thread we are concerned with predicting the function output of the simulator. The prediction of derivatives of the simulator output w.r.t the inputs, at a new point in the input space is covered in the thread *ThreadGenericEmulateDerivatives*. The process of predicting function output at one or more new points for the core problem is set out in *ProcPredictGP*. When we have derivatives in the training sample the process of prediction is the same as for the core problem, but anywhere D, t, A, e etc are required, they should be replaced with $\tilde{D}, \tilde{t}, \tilde{A}, \tilde{e}$.

For some of the tasks considered below, we require to predict the output not at a set of discrete points, but in effect the entire output function as the inputs vary over some range. This can be achieved also using simulation, as discussed in the procedure page for simulating realisations of an emulator (*ProcSimulationBasedInference*).

Uncertainty analysis

Uncertainty analysis is the process of predicting the simulator output when one or more of the inputs are uncertain. The procedure page on uncertainty analysis using a GP emulator (*ProcUAGP*) explains how this is done for the core problem. We hope to extend this procedure to cover an emulator built with derivative information in a later release of the toolkit.

Sensitivity analysis

In *sensitivity analysis* the objective is to understand how the output responds to changes in individual inputs or groups of inputs. The procedure page *ProcVarSAGP* gives details of carrying out *variance based* sensitivity analysis for the core problem. We hope to extend this procedure to cover an emulator built with derivative information in a later release of the toolkit.

8.21.8 Examples

One dimensional example

8.21.9 Additional Comments, References, and Links

If we are interested in emulating multiple outputs of a simulator, there are various approaches to this discussed in the alternatives page *AltMultipleOutputsApproach*. If the approach chosen is to build a *multivariate GP* emulator and derivatives are available, then they can be included using the methods described in this page combined with the methods described in the thread for the analysis of a simulator with multiple outputs (*ThreadVariantMultipleOutputs*). A variant thread on multiple outputs with derivatives (ThreadVariantMultipleOutputsWithDerivatives) page may be included in a later release of the toolkit.

8.22 Procedure: Adaptive Sampler for Complex Models (ASCM)

8.22.1 Description and Background

This procedure aims at sequentially selecting the design points and updating the information at every stage. The framework used to implement this procedure is the Bayesian decision theory. Natural conjugate priors are assumed for the unknown parameters. The ASCM procedure allows for learning about the parameters and assessing the emulator performance at every stage. It uses the Bayesian optimal design principals in *Optimal design*.

8.22.2 Inputs

- The design size n, the subdesign size n_i, a counter $nn = 0$, initial design D_0 usually a space filling design, the candidate set E of size N, also another space filling design usually chosen to be a grid defined on the design space $\mathcal{X}$.
- The output at the initial design points, $Y(D_0)$.
- A model for the output, for simplicity, the model is chosen to be $Y(x) = X\theta + Z(x)$.
- A covariance function associated with the Gaussian process model. In the Karhunen-Loeve method the "true" covariance function is replaced by a truncated version of the K-L expansion *DiscKarhunenLoeveExpansion*.
- A prior distribution for the unkonwn model parameters θ, which is taken here to be $N(\mu, V)$.
- An optimality criterion.
- Note that Σ_{nn} is within-design process covariance matrix and Σ_{nr} the between design and non-design process covariance matrix, and similarly, in the case of unknown σ^2, $\Sigma_{nn} = \sigma^2 R_{nn}$ and $\Sigma_{nr} = \sigma^2 R_{nr}$ where R_{nn}, R_{nr} are the correlation matrices.

8.22.3 Outputs

- A sequentially chosen optimal design D.
- The value of the chosen criterion $\mathcal{C}$.
- The posterior distribution $\pi(\Theta|Y_n)$ and the predictive distribution $f(Y_r|Y_n)$, where Y_n is the vector on n observed outputs and Y_r is the vector of r unobserved outputs. The predictive distribution is used as an emulator.

8.22.4 Procedure

1. Check if the candidate set E contains any points of the initial design points D_0. If it does then $E = E \setminus D_0$.
2. Compute the posterior distribution for the unknown parameters $\pi(\Theta|Y_n)$ and the predictive distribution $f(Y_r|Y_n)$. The posterior distribution can be obtained analytically or numerically.
3. Choose the next design point D_i or points to optimize the chosen criterion. The selection is done using the *exchange algorithm*. The criterion is based on the posterior distribution. For example, the maximum entropy sampling criterion has approximately the form

   $$\det((X_r V X_r^T + \Sigma_{rr} - (X_r V X_n^T + \Sigma_{rn})(X_n V X_n^T + \Sigma_{nn})^{-1}(X_n V X_r^T + \Sigma_{nr})))$$

   if the predictive distribution $f(Y_r|Y_n)$ is a Gaussian process or approximately the form $a^*(X_r V X_r^T + R_{rr} - (X_r V X_n^T + R_{rn})(X_n V X_n^T + R_{nn})^{-1}(X_n V X_r^T + R_{nr}))$ if the predictive distribution is a Student t process. They are almost the same because a^* is just a constant not dependent on the design; the unknown σ^2 does not affect the choice of the design, in this case.
4. Observe the output at the design points D_i selected in step 3. The observation itself is useful for the purposes of assessing the uncertainty about prediction. It is not neccesary for the computing the criterion but it is necessary for computing the predictive distribution $f(Y_r|Y_n)$.
5. Update the predictive distribution $f(Y_r|Y_n)$.
6. Compute the measures of accuracy in order to assess the improvement of prediction.
7. Update the candidate set $E = E \setminus D_i$, the design S, $D = D \cup D_i$, and the design size $nn = nn + n_i$.
8. Stop if $nn = n$ or a certain value of the criterion is achieved stop otherwise go to step 3.

8.22.5 Additional Comments, References, and Links

The above algorithm is a foundation for several other versions under development. The criterion mentioned is the entropy criterion (see step 3 of the procedure), but in principle any other optimal design criterion can be use. The methodology used here allows for learning about the parameters, that is to say the posterior distributions are computed at every design stage, using a basic random regression Bayesian formulation. It also employs the K-L expansion, but any set of basis functions can be used.

The more fully adaptive version, which is under development will allow the variance parameters on the regression terms to be updated, using hyper-parameters. The aim is to allow adaptation to (i) global smoothness, because low/high varying basis function represent less/more smoothness and (ii) local smoothness.

8.23 Procedure: Multivariate lognormal approximation for correlation hyperparameters

8.23.1 Description and Background

The posterior distribution $\pi^*(\delta)$ of the *hyperparameter* vector δ is given in the procedure page for building a Gaussian process emulator for the core problem (*ProcBuildCoreGP*) in the case of the *core problem* (with linear mean function and weak prior information). The ideal way to compute the emulator in this case is to generate a sample of values of δ from this distribution, but that is itself a complex computational task. We present here a simpler method based on a lognormal approximation to $\pi^*(\delta)$.

8.23.2 Inputs

- An emulator as defined in *ProcBuildCoreGP*, using a linear mean and a weak prior.
- The mode of the posterior $\hat{\delta}$ as defined in the discussion page on finding the posterior mode of correlation lengths (*DiscPostModeDelta*).
- The $p \times p$ Hessian matrix $\dfrac{\partial^2 g(\tau)}{\partial \tau^2}$ with (k,l)-th entry $\dfrac{\partial^2 g(\tau)}{\partial \tau_l \partial \tau_k}$, as defined in *DiscPostModeDelta*.

8.23.3 Outputs

- A set of s samples for the correlation lengths δ, denoted as $\tilde{\delta}$.
- A posterior mean $\tilde{m}^*(\cdot)$ and a covariance function $\tilde{u}^*(\cdot,\cdot)$, conditioned on the samples $\tilde{\delta}$

8.23.4 Procedure

- Define $V = -\left(\dfrac{\partial^2 g(\tau)}{\partial \tau^2}\right)^{-1}$. Draw s samples from the p-variate normal distribution $\mathcal{N}(\hat{\tau}, V)$, call these samples $\tilde{\tau}$.
- Calculate the samples $\tilde{\delta}$ as $\tilde{\delta} = \exp(\tilde{\tau}/2)$.
- Given the set of s samples $\tilde{\delta}$, the posterior mean and variance $\tilde{m}^*(\cdot)$, $\tilde{u}^*(\cdot,\cdot)$ can be calculated with the same formulae given in the procedure page for sampling the posterior distribution of the correlation lengths (*ProcMCMCDeltaCoreGP*), or in more detail in the procedure page for predicting the simulator's outputs using a GP emulator (*ProcPredictGP*).

8.23.5 Additional Comments

Most standard statistical computing packages have facilities for taking random samples from a multivariate normal distribution.

When an input is not particularly active, the posterior distribution of the correlation lengths $\pi^*_\delta(\delta)$ can be very flat with respect to that input and obtain its maximum for a large value of δ. This can cause the respective entry of the matrix V to be very large, and the samples $\tilde{\delta}$ that correspond to this input to have both unrealistically large and small values. An inspection of the samples that are returned by the above procedure is recommended, especially in high dimensional input problems, where less active inputs are likely to exist. If the sampled correlation lengths that correspond to one or more inputs are found to have very large (e.g. >50) and very small (e.g. < 0.5) values at the same time, a potential remedy could be to fix the values of these samples to the respective entries of the $\hat{\delta}$ vector.

8.23.6 References

This method is introduced in the following report.

- Nagy B., Loeppky J.L. and Welch W.J. (2007). Fast Bayesian Inference for Gaussian Process Models. Technical Report 230, Department of Statistics, University of British Columbia.

Note however that the authors indicate also that the method works well when the correlation function has the Gaussian form but may not work so well in the case of the exponential power form (see the alternatives page on emulator prior correlation function (*AltCorrelationFunction*):

- Nagy B., Loeppky J.L. and Welch W.J. (2007). Correlation parameterization in random function models to improve normal approximation of the likelihood or posterior. Technical Report 229, Department of Statistics, University of British Columbia.

8.24 Procedure: Iterate the single step emulator using an approximation approach

8.24.1 Description and Background

This page is concerned with task of *emulating* a *dynamic simulator*, as set out in the variant thread for dynamic emulation (*ThreadVariantDynamic*).

We have an emulator for the *single step function* $w_t = f(w_{t-1}, a_t, \phi)$, and wish to predict the full time series $w_1, \ldots, w_T$ for a specified initial *state variable* w_0, time series of *forcing variables* $a_1, \ldots, a_T$ and simulator parameters ϕ. It is not possible to analytically derive a distribution for $w_1, \ldots, w_T$ if $f(\cdot)$ is modelled as a *Gaussian Process*, so here we use an approximation based on the normal distribution to estimate the marginal distribution of each of $w_1, \ldots, w_T$.

8.24.2 Inputs

- An emulator for the single step function $w_t = f(w_{t-1}, a_t, \phi)$, formulated as a GP or *t-process* conditional on hyperparameters, plus a set of hyperparameter values $\theta^{(1)}, \ldots, \theta^{(s)}$.
- An initial value for the state variable w_0.
- The values of the forcing variables $a_1, \ldots, a_T$.
- The values of the simulator parameters ϕ.

8.24.3 Outputs

- Approximate marginal distributions for each of $w_1, \dots, w_T$. The distribution of each w_t is approximated by a normal distribution with a specified mean and variance.

8.24.4 Procedure

For a single choice of emulator hyperparameters θ, we approximate the marginal distribution of w_t by the normal distribution $N_r(\mu_t, V_t)$

We have

$$\begin{aligned}\mu_1 &= m^*(w_0, a_1, \phi),\\ V_1 &= v^*\{(w_0, a_1, \phi), (w_0, a_1, \phi)\}.\end{aligned}$$

The mean and variance are defined recursively:

$$\begin{aligned}\mu_{t+1} &= \mathrm{E}[m^*(w_t, a_{t+1}, \phi)|f(D), \theta],\\ V_{t+1} &= \mathrm{E}[v^*\{(w_t, a_{t+1}, \phi), (w_t, a_{t+1}, \phi)\}|f(D), \theta] + \mathrm{Var}[m^*(w_t, a_{t+1}, \phi)|f(D), \theta],\end{aligned}$$

where the expectations and variances are taken with respect to w_t, where $w_t \sim N_r(\mu_t, V_t)$

Explicit formulae for μ_{t+1} and V_{t+1} can be derived in the case of a linear mean and a separable Gaussian covariance function. The procedure for calculating μ_{t+1} and V_{t+1} is described in *ProcUpdateDynamicMeanAndVariance*. Otherwise, we can use simulation to estimate μ_{t+1} and V_{t+1}. A simulation procedure is given in *ProcApproximateUpdateDynamicMeanandVariance*.

8.24.5 Integrating out the emulator hyperparameters

Assuming we have $s > 1$, we can integrate out the emulator hyperparameters to obtain the unconditional mean and variance of w_t using Monte Carlo estimation. In the following procedure, we define N to be the number of Monte Carlo iterations, and for notational convenience, we suppose that $N \leq s$. For discussion of the choice of N, including the case $N > s$, see the discussion page on Monte Carlo estimation, sample sizes and emulator hyperparameter sets (*DiscMonteCarlo*).

1. For $i = 1, 2, \dots, N$ fix the hyperparameters at the value $\theta^{(i)}$, and calculate the corresponding mean and variance of w_t, which we denote by $\mu_t^{(i)}$ and $V_t^{(i)}$.
2. Estimate $\mathrm{E}[w_t|f(D)]$ by

$$\hat{E}_t = \frac{1}{N}\sum_{i=1}^{N}\mu_t^{(i)}.$$

3. Estimate $\mathrm{Var}[w_t|f(D)]$ by

$$\frac{1}{N}\sum_{i=1}^{N}V_t^{(i)} + \frac{1}{N-1}\sum_{i=1}^{N}\left(\mu_t^{(i)} - \hat{E}_t\right)^2.$$

8.25 Procedure: Use simulation to recursively update the dynamic emulator mean and variance in the approximation method

8.25.1 Description and Background

This page is concerned with task of *emulating* a *dynamic simulator*, as set out in the variant thread on dynamic emulation (*ThreadVariantDynamic*).

The approximation procedure for iterating the single step emulator (*ProcApproximateIterateSingleStepEmulator*) recursively defines

$$\begin{aligned}\mu_{t+1} &= \mathrm{E}[m^*(w_t, a_{t+1}, \phi)|f(D)],\\ V_{t+1} &= \mathrm{E}[v^*\{(w_t, a_{t+1}, \phi), (w_t, a_{t+1}, \phi)\}|f(D)] + \mathrm{Var}[m^*(w_t, a_{t+1}, \phi)|f(D)],\end{aligned}$$

where the expectations and variances are taken with respect to w_t, with $w_t \sim N_r(\mu_t, V_t)$. If the *single step* emulator has a linear mean and a separable Gaussian covariance function, then μ_{t+1} and V_{t+1} can be computed explicitly, as described in the procedure page for recursively updating the dynamic emulator mean and variance (*ProcUpdateDynamicMeanAndVariance*). Otherwise, simulation can be used, which we describe here.

8.25.2 Inputs

- μ_t and V_t
- The single step emulator, conditioned on training inputs D, and hyperparameters θ, with posterior mean and covariance functions $m^*(\cdot)$ and $v^*(\cdot, \cdot)$ respectively.

8.25.3 Outputs

- Estimates of μ_{t+1} and V_{t+1}

8.25.4 Procedure

We describe a Monte Carlo procedure using N Monte Carlo iterations. For discussion of the choice of N, see the discussion page on Monte Carlo estimation (*DiscMonteCarlo*).

1. For $i = 1, \ldots, N$, sample $w_t^{(i)}$ from $N(\mu_t, V_t)$
2. Estimate μ_{t+1} by $\hat{\mu}_{t+1} = \frac{1}{N}\sum_{i=1}^N m^*(w_t^{(i)}, a_{t+1}, \phi)$
3. Estimate V_{t+1} by

$$\hat{V}_{t+1} = \frac{1}{N}\sum_{i=1}^N v^*\{(w_t^{(i)}, a_{t+1}, \phi), (w_t^{(i)}, a_{t+1}, \phi)\} + \frac{1}{N-1}\sum_{i=1}^N \left(m^*(w_t^{(i)}, a_{t+1}, \phi) - \hat{\mu}_{t+1}\right)^2$$

8.26 Procedure: Automatic Relevance Determination (ARD)

We describe here the method of Automatic Relevance Determination (ARD) where the *correlation length scales* δ_i in a covariance function can be used to determine the input relevance. This is also known as the application of independent priors over the length scales in the covariance models.

The purpose of the procedure is to perform *screening* on the simulator inputs, identifying the *active* inputs.

ARD is typically applied using a zero mean *Gaussian Process emulator*. Provided the inputs have been standardised (see the procedure page on data preprocessing (*ProcDataPreProcessing*)), the correlation length scales may be directly used as importance measures. Another case where ARD may be used is with non-zero mean function Gaussian Process where we wish to identify factor effects in the residual process. For example with a linear mean, correlation length scales indicate non-linear and interaction effects. If the effect of a factor is strictly linear with no interaction with other factors, it can still be screened out by subtracting from the simulator output prior to emulation.

8.26.1 Choice of Covariance Function

To implement the ARD method, a range of covariance functions can be used. In fact any covariance function that has a length scale vector included can be used for ARD, for example the squared exponential covariance used in most of the toolkit. Such a covariance function is the Rational Quadratic (RQ) :

$v(x_p, x_q) = \sigma^2[1 + (x_p - x_q)^\tau P^{-1}(x_p - x_q)/(2\alpha)]^{-\alpha}$ where σ is the scale parameter and $P = \mathrm{diag}(\delta_i)^2$ a diagonal matrix of correlation length scale parameters. Taking the limit $a \to \infty$ parameter, we obtain the squared exponential kernel.

Assuming p input variables, each hyperparameter δ_i is associated with a single input factor. The δ_i hyperparameters are referred to as characteristic length scales and can be interpreted as the distance required to move along a particular axis for the function values to become uncorrelated. If the length-scale has a very large value the covariance becomes almost independent of that input, effectively removing that input from the model. Thus length scales can be viewed as a total effect measure and used to determine the relevance of a particular input.

Lastly, if the simulator produces random outputs the emulator should no longer exactly interpolate the observations. In this case, a nugget term ν should be added to the covariance function to capture the response uncertainty.

8.26.2 Implementation

Given a set of simulator runs, the ARD procedure can be implemented in the following order:

1. **Standardisation**. It is important to first *standardise* the input data so all input factors operate on the same scale. If rescaling is not done prior to the inference stage, length scale parameters will generally have larger values for input factors operating on larger scales.
2. **Inference**. The Maximum-A-Posteriori (MAP) values of the length scale hyper-parameters are typically obtained by iterative non-linear optimisation using standard algorithms such as scaled conjugate gradients, although in a fully Bayesian treatment posterior distributions could be approximated using Monte Carlo methods. Maximum-A-Posteriori (MAP) is the process of identifying the mode of the posterior distribution of the hyperparameter (see the discussion page *DiscPostModeDelta*). One difficulty using ARD stems from the use of an optimisation process since the optimisation is not guaranteed to converge to a global minimum and thus ensure robustness. The algorithm can be run multiple times from different starting points to assess robustness at the cost of increasing the computational resources required. In case of a very high dimensional input space, maximum likelihood may be too costly or intractable due to the high number of free parameters (one length scale for each dimension). In this case Welch at al (1992) propose a constrained version of maximum likelihood where initially all inputs are assumed to have the same length scale and iteratively, some inputs are assigned separate length scales based on the improvement in the likelihood score.
3. **Validation**. To ensure robustness of the screening results, prior to utilising the length scales as importance measures the emulator should be validated as described in procedure page *ProcValidateCoreGP*.

An example of applying the ARD process is provided in *ExamScreeningAutomaticRelevanceDetermination*.

8.26.3 References

Williams, C. K. I. and C. E. Rasmussen (2006). Gaussian Processes for Machine Learning. MIT Press.

William J. Welch, Robert. J. Buck, Jerome Sacks, Henry P. Wynn, Toby J. Mitchell and Max D. Morris. " Screening, Predicting, and Computer Experiments", *Technometrics*, Vol. 34, No. 1 (Feb., 1992), pp. 15-25. Available at http://www.jstor.org/stable/1269548.

8.27 Procedure: Calculation of adjusted expectation and variance

8.27.1 Description and Background

In the context of *Bayes linear* methods, the Bayes linear *adjustment* is the appropriate method for updating prior *second-order beliefs* given observed data. The adjustment takes the form of linear fitting of our beliefs on the observed data quantities. Specifically, given two random vectors, B, D, the *adjusted expectation* for element B_i, given D, is the linear combination $a_0 + a^T D$ minimising $\mathrm{E}[B_i - a_0 - a^T D)^2]$ over choices of $\{a_0, a\}$.

8.27.2 Inputs

- $\mathrm{E}[B]$, $\mathrm{Var}[B]$ - prior expectation and variance for the vector B
- $\mathrm{E}[D]$, $\mathrm{Var}[D]$ - prior expectation and variance for the vector D
- $\mathrm{Cov}[B, D]$ - prior covariance between the vector B and the vector B
- D_{obs} - observed values of the vector D

8.27.3 Outputs

- $\mathrm{E}_D[B]$ - adjusted expectation for the uncertain quantity B given the observations D
- $\mathrm{Var}_D[B]$- adjusted variance matrix for the uncertain quantity B given the observations D

8.27.4 Procedure

The adjusted expectation vector, $\mathrm{E}_D[B]$ is evaluated as

$$\mathrm{E}_D[B] = \mathrm{E}[B] + \mathrm{Cov}[B, D]\mathrm{Var}[D]^{-1}(D_{obs} - \mathrm{E}[D])$$

(If $\mathrm{Var}[D]$ is not invertible, then we use a generalised inverse such as Moore-Penrose).

The *adjusted variance matrix* for B given D is

$$\mathrm{Var}_D[B] = \mathrm{Var}[B] - \mathrm{Cov}[B, D]\mathrm{Var}[D]^{-1}\mathrm{Cov}[D, B]$$

8.27.5 Additional Comments

See *DiscBayesLinearTheory* for a full description of Bayes linear methods.

8.28 Procedure: Predict simulator outputs using a BL emulator

8.28.1 Description and Background

A *Bayes linear* (BL) *emulator* is a stochastic representation of knowledge about the outputs of a *simulator* based on a *second-order belief specification* for an unknown function. The unknown function in this case is the simulator, viewed as a function that takes inputs and produces one or more outputs. One use for the emulator is to predict what the simulator would produce as output when run at one or several different points in the input space. This procedure describes how to derive such predictions in the case of a BL emulator such as is produced by *ProcBuildCoreBL*.

8.28.2 Inputs

- An adjusted Bayes linear emulator
- A single point x' or a set of points $x'_1, x'_2, \dots, x'_{n'}$ at which predictions are required for the simulator output(s)

8.28.3 Outputs

- In the case of a single point, outputs are the adjusted expectation and variance at that point
- In the case of a set of points, outputs are the adjusted expectation vector and adjusted variance matrix for that set

8.28.4 Procedure

The adjusted Bayes linear emulator will supply the following necessary pieces of information:

- An adjusted expectation $\mathrm{E}_F[\beta]$ and variance $\mathrm{Var}_F[\beta]$ for the trend coefficients β given the model runs F
- An adjusted expectation $\mathrm{E}_F[w(x)]$ and variance $\mathrm{Var}_F[w(x)]$ for the residual process $w(x)$, at any point x, given the model runs F
- An adjusted covariance $\mathrm{Cov}_F[\beta, w(x)]$ between the trend coefficients and the residual process

The adjusted expectation and variance at the new point x' are obtained by application of *ProcBLAdjust* to the emulator as described below.

Predictive mean (vector)

Then our adjusted beliefs about the expected simulator output at a single further input configuration x' are given by:

$$\mathrm{E}_F[f(x')] = h(x')^T \mathrm{E}_F[\beta] + \mathrm{E}_F[w(x').$$

In the case of a set of additional inputs X', where X' is the matrix with rows $x'_1, x'_2, \dots, x'_{n'}$, the adjusted expectation is:

$$\mathrm{E}_F[f(X')] = H(X')^T \mathrm{E}_F[\beta] + \mathrm{E}_F[w(X')$$

where $f(X)$ is the n'-vector of simulator values with elements $(f(x'_1), f(x'_2), \dots, f(x'_{n'}))$, $H(X')$ is the $n' \times q$ matrix with rows $h(x'_1), h(x'_2), \dots, h(x'_{n'})$, and $w(X)$ is the n'-vector with elements $(w(x'_1), w(x'_2), \dots, w(x'_{n'}))$.

Predictive variance (matrix)

Our adjusted variance of the simulator output at a single further input configuration x' is given by:

$$\text{Var}_F[f(x')] = h(x')^T \text{Var}_F[\beta] h(x') + \text{Var}_F[w(x')] + 2h(x')^T \text{Cov}_F[\beta, w(x')]$$

In the case of a set of additional inputs X', the adjusted variance is:

$$\begin{aligned}\text{Var}_F[f(X')] =& H(X')^T \text{Var}_F[\beta] H(X') + \text{Var}_F[w(X')] + \\ & H(X')^T \text{Cov}_F[\beta, w(X')] + \text{Cov}_F[w(X'), \beta] H(X').\end{aligned}$$

8.29 Procedure: Bayes linear method for learning about the emulator residual variance

8.29.1 Description and Background

This page assumes that the reader is familiar with the concepts of *exchangeability*, *second-order exchangeability*, and the general methodology of Bayes linear belief adjustment for collections of exchangeable quantities as discussed in *DiscAdjustExchBeliefs*.

In the study of computer models, learning about the mean behaviour of the simulator can be a challenging task though there are many tools available for such an analysis. The problem of learning about the variance of a simulator, or the variance parameters of its emulator is more difficult still. One approach is to apply the Bayes linear methods for adjusting exchangeable beliefs. For example, suppose our emulator has the form

$$f(x) = \sum_j \beta_j h_j(x) + e(x),$$

then if our trend component is known and the residuals can be treated as second-order exchangeable then we can directly apply the methods of *DiscAdjustExchBeliefs* to learn about $\text{Var}[e(x)]$, the variance of the residual process. The constraint on this procedure is that we require the emulator residuals to be (approximately) second-order exchangeable. In the context of computer models, we can reasonably make this judgement in two situations:

1. **The design points $D = (x_1, \ldots, x_n)^T$ are such that they are sufficiently well-separated that they can be treated as uncorrelated.** This may occur due to simply having a small number of points, or having a large number of points in a high-dimensional input space. In either case, the relative sparsity of model evaluations means that we can express the covariance matrix of the emulator residuals as $\text{Var}[e(D)] = \sigma^2 I_n$ where I_n is the $n \times n$ identity matrix and only σ^2 is uncertain. In this case, the residuals $e(x)$ have the same prior mean (0), the same prior variance (σ^2) and every pair of residuals has the same covariance (0) therefore the residuals can be treated as second-order exchangeable.
2. **The form of the correlation function $c(x, x')$ and its parameters δ are known.** In this case, for any number or arrangement of design points we can express $\text{Var}[e(D)] = \sigma^2 R$ where R is a known matrix of correlations, and σ^2 is the unknown variance of the residual process. Since the correlation between every pair of residuals is not constant, we do not immediately have second-order exchangeability. However, since the form of R is known we can perform a linear transformation of the residuals which will preserve the constant mean and variance but will result in a correlation matrix of the form $R = I_n$ and so resulting in a collection of transformed second-order exchangeable residuals which can be used to learn about σ^2.

The general process for the adjustment is identical for either situation, however in the second case we must perform a preliminary transformation step.

8.29.2 Inputs

The following inputs are required:

- Output vector $f(D) = (f(x_1), \dots, f(x_n))$, where $f(x_i)$ is the scalar simulator output corresponding to input vector x_i in D;
- The form of the emulator trend basis functions $h_j(\cdot)$
- Design matrix $D = (x_1, \dots, x_n)$.
- Specification of prior beliefs for $\omega_e = \sigma^2$ and the fourth-order quantities $\omega_{\mathcal{M}}$ and $\omega_{\mathcal{R}}$ as defined below and in *DiscAdjustExchBeliefs*.

We also make the following requirements:

- **Either**: The design points in D are sufficiently well-separated that the corresponding residuals can be considered to be uncorrelated
- **Or**: The form of the correlation function, $c(x, x')$, and the values of its hyperparameters, δ, (and hence the correlations between every pair of residuals) are known.

8.29.3 Outputs

- Adjusted expectation and variance for the variance of the residual process

8.29.4 Procedure

Overview

Our goal is to learn about the variance of the residual process, $e(x)$, in the emulator of a given computer model. Typically, we assume that our emulator has a mean function which takes linear form in some appropriate basis functions of the inputs, and so we express our emulator as

$$f(x) = \beta^T h(x) + e(x),$$

where β is a vector of emulator trend coefficients, $h(x)$ is a vector of the *basis functions* evaluated at input x, and $e(x)$ is a stochastic residual process. We consider that the coefficients $\beta = (\beta_1, \dots, \beta_q)$ are unknown, and then work with the derived residual quantities $e_i = e(x_i)$. We assume that we consider the residual process to be weakly stationary with mean zero a priori, which gives

$$\begin{aligned} e_i &= f(x_i) - \beta_1 h_1(x_i) - \cdots - \beta_q h_q(x_i) \\ \mathrm{E}[e_i] &= 0 \\ \mathrm{Var}[e_i] &= \sigma^2 = \omega_e \end{aligned}$$

where we introduce $\omega_e = \sigma^2$ as the variance of $e(x)$ for notational convenience and to mirror the notation of *DiscAdjustExchBeliefs*.

Orthogonalisation

In the case where the emulator residuals are not uncorrelated, but can be expressed in the form $\mathrm{Var}[e] = \sigma^2 R$, where R is a known $n \times n$ correlation matrix, we are required to make a transformation in order to de-correlate the residuals in order to obtain a collection of second-order exchangeable random quantities. To do this, we adopt the standard approach in regression with correlated errors – namely generalised least squares.

Let Q be any matrix satisfying $QQ^T = R$, and we can then transform the emulator $f(D) = X\beta + e$ to the form

$$f'(D) = X'\beta + e',$$

where $f'(D) = Q^{-1}f(D)$, $X' = Q^{-1}Z$, and $e' = Q^{-1}e$. An example of a suitable matrix Q would be if we find the eigen-decomposition of R such that $R = A\Lambda A^T$ then $Q^{-1} = \Lambda^{-\frac{1}{2}}A^T$ would provide a suitable transformation matrix. Under this transformation, we have that

$$\begin{aligned}\mathrm{E}[e'] &= Q^{-1}\mathrm{E}[e] = 0\\ \mathrm{Var}[e'] &= Q^{-1}\mathrm{Var}[e]Q^{-T} = \omega_e I_n.\end{aligned}$$

Note that he transformed residuals e' have both the same mean and variance as the un-transformed residuals e_i, and in particular note that $\mathrm{Var}[e_i] = \mathrm{Var}[e'_i] = \sigma^2$ which is the quantity we seek to estimate. Further, the transformed residuals e' are second-order exchangeable as they have a common mean and variance, and every pair has a common covariance.

Exchangeability Representation

In order to revise our beliefs about the population residual variance, we judge that the residuals e_i are second-order exchangeable. When the residuals are well-separated and uncorrelated, this is immediately true. In the case of the known correlations, then we make this statement about the transformed residuals, e'_i, and proceed through the subsequent stages operating with the e'_i instead of e_i. For simplicity, from this point on we only discuss e_i and assume that any necessary orthogonalisation has been made.

We begin with the uncorrelated second-order exchangeable sequence of residuals e_i. Suppose further that we judge that the e_i^2 are also second-order exchangeable and so we write

$$v_i = e_i^2 = \mathcal{M}(v) + \mathcal{R}_i(v)$$

where $\mathrm{E}[\mathcal{M}(v)] = \omega_e = \sigma^2$, $\mathrm{Var}[\mathcal{M}(v)] = \omega_{\mathcal{M}}$, and that the $\mathcal{R}_i(v)$ are SOE, uncorrelated and have zero mean and variance $\mathrm{Var}[\mathcal{R}_i(v)] = \omega_{\mathcal{R}}$. We also make the fourth-order uncorrelated assumptions mentioned in *DiscAdjustExchBeliefs*.

In order to adjust our beliefs about the population residual variance, we use the residual mean square $\hat{\sigma}^2$,

$$\hat{\sigma}^2 = \frac{1}{n-q}\hat{e}^T\hat{e},$$

where $\hat{e} = f(D) - X\hat{\beta} = (I_n - H)f(D)$, where H is the idempotent matrix $H = X(X^TX)^{-1}X^T$, X is the model matrix with i-th row equal to $(h_1(x_i), \ldots, h_q(x_i))$, and $\hat{\beta}$ are the least-squares estimates for β given by $\hat{\beta} = (X^TX)^{-1}X^Tf(D)$. We could update our beliefs by other quantities, though s^2 has a relatively simple representation improving the tractability of subsequent calculations.

We can now express $\hat{\sigma}^2$ as

$$\hat{\sigma}^2 = \mathcal{M}(v) + T,$$

and

$$T = \frac{1}{n-q}\left[\sum_k (1-h_{kk})\mathcal{R}_k(v) - 2\sum_{k<j} h_{kj}e_ke_j\right]$$

and it follows that we have the follow belief statements

$$\begin{aligned}\mathrm{E}[\hat{\sigma}^2] &= \omega_e = \sigma^2,\\ \mathrm{Var}[\hat{\sigma}^2] &= \omega_{\mathcal{M}} + \omega_t,\\ \mathrm{Cov}[\hat{\sigma}^2, \mathcal{M}(v)] &= \omega_{\mathcal{M}},\\ \omega_T &= \frac{1}{(n-q)^2}\left[\omega_{\mathcal{R}}\sum_k (1-h_{kk})^2 - 2(\omega_{\mathcal{M}} + \omega_e^2)\sum_k h_{kk}^2 + 2q(\omega_{\mathcal{M}} + \omega_e^2)\right],\end{aligned}$$

which complete our belief specification for $\hat{\sigma}^2$ and $\mathcal{M}(v)$.

Variance Adjustment

Given the beliefs derived as above and the residual mean square $\hat{\sigma}^2$ as calculated from the emulator runs and emulator trend, we obtain the following expression for the adjusted mean and variance for $\mathcal{M}(v)$, the population residual variance:

$$\begin{aligned}\mathrm{E}_{\hat{\sigma}^2}[\mathcal{M}(v)] &= \frac{\omega_{\mathcal{M}}\hat{\sigma}^2 + \omega_T\omega_e}{\omega_{\mathcal{M}} + \omega_T} \\ \mathrm{Var}_{\hat{\sigma}^2}[\mathcal{M}(v)] &= \frac{\omega_{\mathcal{M}}\omega_t}{\omega_{\mathcal{M}} + \omega_t}\end{aligned}$$

8.29.5 Comments and Discussion

When approaching problems based on exchangeable observations, we are often also interested in learning about the population mean in addition to the population variance. In terms of computer models, the primary goal is to learn about the mean behaviour of our emulator residuals rather than the emulator variance. To combine these approaches, we carry out the analysis in two stages. For the first stage, we carry out variance assessement as described above which gives us a revised estimate for our residual variance, σ^2. In the second stage, we perform the standard Bayes linear analysis for the mean vector. This involves following the standard methods of learning about the emulator residual means as described in *ProcBuildCoreBL*, having replaced our prior value for the residual variance with the adjusted estimate obtained from the methods above. This procedure is called a **two-stage Bayes linear analysis**, and is a simpler alternative to jointly learning about both mean and variance which ignores uncertainty in the variance when updating the mean vector.

8.29.6 References

- Goldstein, M. and Wooff, D. A. (2007), Bayes Linear Statistics: Theory and Methods, Wiley.

8.30 Procedure: Branch and bound algorithm

8.30.1 Description and Background

Branch and Bound algorithm is a commonly used technique in several mathematical programming applications especially in combinatorial problem when a problem is difficult to be solved directly. It is preferred over many other algorithms because it reduces the amount of search needed to find the optimal solution.

Branch and Bound creates a set of subproblems by dividing the space of current problem into unexplored subspaces represented as nodes in a dynamically generated search tree, which initially contains the root (the original problem). Performing one iteration of this procedure is based on three main components: selection of the node to process, bound calculation, and branching, where branching is the partitioning process at each node of the tree and bounding means finding lower and upper bounds to construct a proof of optimality without exhaustive research. The algorithm can be terminated whenever the difference between the upper bound and the lower bound is smaller than the chosen ϵ or if the set of live nodes is empty, i.e. there is no unexplored parts of the solution space left, and the optimal solution is then the one recorded as “current best”.

The branch and bound procedure explained here is for finding a maximum entropy design. The computations are based on the process covariance matrix of a large candidate set of order $N \times N$, stored as a function. Following the Maximum Entropy Sampling principle, the goal is to find the design of sample size n whose $n \times n$ process covariance matrix has maximum determinant. Since the design is a subset of the candidate set the design covariance matrix will be a submatrix of the candidate set covariance matrix.

8.30.2 Inputs

1. Candidate set of N points
2. A known covariance matrix (stored as a function) of the candidate points
3. E the set of all eligible points
4. F the set of all points forced to be in the design
5. $\epsilon > 0$ small chosen number
6. A counter $k = 0$
7. An initial design of size S_0 of size n
8. An optimality criterion $I = \log\det C[S]$ (the version described is for entropy).
9. General upper and lower bounds: $U = Ub(C, F, E, s)$, $L = Lb(C, F, E, s)$.

8.30.3 Outputs

1. Global optimal design for the problem
2. Value of the objective
3. Various counts such as number of branchings made

8.30.4 Procedure

1. Let k stand for the iteration number, U_k stand for the upper bound at k th iteration, I, the incumbent, i.e. for best current value of the target function and L be the set of all unexplored subspaces
2. Remove the problem, tuple, with max Ub from L in order to be explored.
3. Branch the problem according to these conditions
4. Set $k = k + 1$
   - If $|F|+|E|-1 > s$, compute $Ub(C, F, E\backslash i, s)$ and if $Ub(C, F, E\backslash i, s) > U_k$, where i is any index selected to be removed from E, then add $(C, F, E \setminus i, s)$ to L, else if $|F| + |E| - 1 = s$, then set $S = F \bigcup E \setminus i$ and compute $\log\det Cov[S]$, and if $\log\det Cov[S] > I$ then set $I = \log\det Cov[S]$ and the current design is S.
   - If $|F|+1 < s$, compute $Ub(C, F \bigcup i, E \setminus i, s)$ and if $Ub(C, F \bigcup i, E \setminus i, s) > U_k$, then add $(C, F, E \setminus i, s)$ to L else if $|F| + 1 < s$, then set $S = F \bigcup i$ and compute $\log\det S$ and if $\log\det Cov[S] > I$ then set $I = \log\det S$ and the current design is S.
5. Update U_{k+1} to be the highest upper bound in L.
6. Repeat steps 3,4,5 while $U_k - I > \epsilon$.

8.30.5 Additional Comments, References, and Links

The upper bound used in the algorithm above are based on spectral bounds for determinants: the determinant of a non-negative definite matrix (eg $C[S]$) is less than or equal to the product of the diagonal elements, which is taken as the upper bound. In this version of the branch and bound the current value of the objective function is taken as the lower bound.

In principle the same algorithm can easily be applied to other optimal design criteria if a general upper bounding function can be found. For example for IMSEP (prediction MSE) we want to *minimise* the objective and need a lower bound. The maximum eigen-value of the posterior covariance matrix of the unsampled point in the candidate set can be used, although expensive to compute.

The following is a main reference on branch and bound for maximum entropy sampling.

W. K. Chun, J. Lee, and M. Queyranne. An exact algorithm for maximum entropy sampling. *Oper. Res.*, 43(4):684-691, 1995

A general reference on combinatorial optimisation which contains useful material on the Branch and Bound algorithm is:

Handbook of combinatorial optimization. Supplement Vol. B. Edited by Ding-Zhu Du and Panos M. Pardalos. Springer-Verlag, New York, 2005.

8.31 Procedure: Building a Bayes linear emulator for the core problem (variance parameters known)

8.31.1 Description and Background

The preparation for building a *Bayes linear* (BL) *emulator* for the *core problem* involves defining the prior mean and covariance functions, specifying prior expectations and variances for the *hyperparameters*, creating a *design* for the *training sample*, then running the *simulator* at the input configurations specified in the design. All of this is described in the thread for Bayes linear emulation for the core model (*ThreadCoreBL*). In this case, we consider taking those various ingredients with specified point values for the variance parameters and creating the BL emulator.

8.31.2 Inputs

- Basis functions, $h(\cdot)$ for the prior mean function $m_\beta(\cdot)$
- Prior expectation, variance and covariance specifications for the regression coefficients β
- Prior expectation for the residual process $w(x)$
- Prior covariance between the coefficients and the residual process
- Specified correlation form for $c_\delta(x, x')$
- Specified values for σ^2 and δ
- Design X comprising points $\{x_1, x_2, \ldots, x_n\}$ in the input space
- Output vector $F = (f(x_1), f(x_2), \ldots, f(x_n))^T$, where $f(x_j)$ is the simulator output from input vector x_j

8.31.3 Outputs

- Adjusted expectations, variances and covariances for β
- Adjusted residual process
- Adjusted covariance between β and the residual process

These outputs, combined with the form of the mean and covariance functions, define the emulator and allow all necessary computations for tasks such as prediction of the simulator output, *uncertainty analysis* or *sensitivity analysis*.

8.31.4 Procedure

The procedure of building the Bayes linear emulator is simply the *adjustment* (as described in the procedure page on calculating the adjusted expectation and variance (*ProcBLAdjust*)) of the emulator by the observed simulator outputs.

Adjustment

To adjust our beliefs for β and the residual process $w(x)$ we require the following prior specifications:

- $\mathrm{E}[\beta]$, $\mathrm{Var}[\beta]$ - prior expectation and variance for the regression coefficients β
- $\mathrm{E}[w(x)]$, $\mathrm{Var}[w(x)]$ - prior expectation and variance for the residual process $w(\cdot)$ at any point x in the input space
- $\mathrm{Cov}[w(x), w(x')]$ - prior covariance between the residual process $w(\cdot)$ at any pair of points (x, x')
- $\mathrm{Cov}[\beta, w(x)]$ - prior covariance between the regression coefficients β and the residual process $w(\cdot)$ at any point x

Given the relationship $f(x) = h(x)^T\beta + w(x)$, define the following quantities obtained from the prior specifications:

Adjusted expectation and variance for trend coefficients

Define $H(X)$ to be the $n \times q$ matrix of basis functions over the design with rows $h(x_1), h(x_2), \ldots, h(x_n)$, and $w(X)$ to be the n-vector of emulator trend residuals with elements $w(x_1), w(x_2), \ldots, w(x_n)$ where x_i is the i-th point in the design X. Then the adjusted expectation and variance for β are given by:

$$\begin{aligned}
\mathrm{E}_F[\beta] &= \mathrm{E}[\beta] + \mathrm{Var}[\beta]H(X)\{H(X)^T\mathrm{Var}[\beta]H(X) + \mathrm{Var}[w(X)]\}^{-1} \times (F - H(X)^T\mathrm{E}[\beta] - \mathrm{E}[w(X)]) \\
\mathrm{Var}_F[\beta] &= \mathrm{Var}[\beta] - (\mathrm{Var}[\beta]H(X))\{H(X)^T\mathrm{Var}[\beta]H(X) + \mathrm{Var}[w(X)]\}^{-1}(H(X)^T\mathrm{Var}[\beta])
\end{aligned}$$

Adjusted expectation and variance for residual process

The adjusted expectation and variance for $w(\cdot)$ at any further input point x, and the adjusted covariance between any further pair of points (x, x') are given by:

$$\begin{aligned}
\mathrm{E}_F[w(x)] &= \mathrm{E}[w(x)] + \mathrm{Cov}[w(x), w(X)]\{H(X)^T\mathrm{Var}[\beta]H(X) + \mathrm{Var}[w(X)]\}^{-1} \times (F - H(X)^T\mathrm{E}[\beta] - \mathrm{E}[w(X)]) \\
\mathrm{Var}_F[w(x)] &= \mathrm{Var}[w(x)] - \mathrm{Cov}[w(x), w(X)]\{H(X)^T\mathrm{Var}[\beta]H(X) + \mathrm{Var}[w(X)]\}^{-1}\mathrm{Cov}[w(X), w(x)] \\
\mathrm{Cov}_F[w(x), w(x')] &= \mathrm{Cov}[w(x), w(x')] - \mathrm{Cov}[w(x), w(X)]\{H(X)^T\mathrm{Var}[\beta]H(X) + \mathrm{Var}[w(X)]\}^{-1}\mathrm{Cov}[w(X), w(x')]
\end{aligned}$$

Adjusted covariance between trend coefficients and residual process

The adjusted covariance between the trend coefficients and the residual process $w(\cdot)$ at any further input point x is given by:

$$\mathrm{Cov}_F[\beta, w(x)] = \mathrm{Cov}[\beta, w(x)] - \mathrm{Var}[\beta]H(X)\{H(X)^T\mathrm{Var}[\beta]H(X) + \mathrm{Var}[w(X)]\}^{-1}\mathrm{Cov}[w(X), w(x)]$$

8.32 Procedure: Empirical construction of a Bayes linear emulator for the core problem using only simulator runs

8.32.1 Description and Background

Ordinarily, when constructing *Bayes linear emulators* we rely on prior information to direct the choice of *emulator* structure and we combine prior beliefs about the emulator parameters with model evaluations to generate our emulator following the methods of the procedure on building a Bayes linear emulator for the core problem (*ProcBuildCoreBL*). However, when the *computer model* is relatively inexpensive to evaluate and prior information is comparatively limited, then emulator choice may be made on the basis of a very large collection of *simulator evaluations*. In particular, this approach is used for the emulation of a fast approximate version of the simulator (see *multi-scale emulation*).

In this situation, we may make many runs of the simulator, allowing us to develop a preliminary view of the form of the function, and thereby to make preliminary choices of the *basis function* collection $\{h_j(x)\}$, and suggest an informed prior specification for the random quantities that determine the emulator for the simulator $f(x)$. This analysis is supported by a diagnostic analysis, for example based on looking for systematic structure in the emulator residuals.

With such a large number of evaluations of the model, the emulator's global trend can be identified and well-estimated from the data alone without application of *Bayes*. For a Bayesian treatment at this stage, our prior judgements would be dominated by the large number of model evaluations so typically we use standard model-fitting techniques.

As we are assuming that we have no substantial prior information about the emulators, then we would typically consider evaluating the computer model over a space-filling design over the input space to obtain good coverage and to learn about global variation in $f(x)$. We assume we start this procedure with a large design (generated by methods such as those discussed in the alternatives page on training sample design for the core problem (*AltCoreDesign*)), and the corresponding simulator evaluations at each of these input parameter combinations.

8.32.2 Inputs

- Design D over the input space comprising the input points $\{x_1, x_2, \dots, x_n\}$
- Output vector $f(D) = (f(x_1), f(x_2), \dots, f(x_n))^T$, where $f(x_j)$ is the simulator output corresponding to input vector x_j in D
- A collection of potential basis functions, $h(\cdot)$, for the prior mean function $m(\cdot)$

8.32.3 Outputs

- A collection of appropriate basis functions for the emulator mean function
- Expectation vector and variance/covariance matrix for regression coefficients β
- Adjusted residual process and specification of covariance function hyperparameters (σ^2, δ)

8.32.4 Procedure

The general process for the empirical construction of an emulator proceeds in the following four stages:

1. Determine the *active inputs* to the emulator
2. Determine an appropriate subset of basis functions
3. Estimate emulator regression coefficients β
4. Estimate residual process hyperparameters (σ^2, δ)

Determine active inputs

If we have chosen to work with *active inputs* in the means function, then the first step in constructing the emulator is to identify the subset of inputs, x_A, which drive the majority of global variation in $f(x)$. For this stage, we require a set of possible basis functions (see the alternatives page on basis functions for the emulator mean (*AltBasisFunctions*) for details.) Given this set of possible regressors and the ample supply of computer evaluations, we can determine the important inputs and model effects by methods such as stepwise fitting.

There are many possible approaches for empirically determining active variables. The process of identifying active variable is known as *screening*. Typically, these methods take the form of model selection and model search problems. A simple such approach using backward stepwise regression would be:

1. Fit the emulator mean function using all possible basis functions - this is now the ‘current’ model
2. For each input in the current model, remove all terms involving that input variable and re-fit the mean function
3. Compare each of these sub-models with the current model using an appropriate criterion
4. The most favourable sub-model now becomes the current model
5. Iterate until an appropriate stopping criterion is satisfied

When the input space is very high-dimensional, a backward stepwise approach may not be viable due to the large number of possible terms in the initial mean function. In these cases, forward selection approaches would be more appropriate beginning with a simple constant as the initial mean function and adding terms in active variables at each stage rather than removing them. Given a very large collection of potential inputs, where possible, it is helpful to start the stepwise search with a sub-collection of input suggested by expert knowledge of the physical processes. Other approaches to *screening* are discussed in the topic thread on screening (*ThreadTopicScreening*).

Determine regression basis functions

The procedure for determining an appropriate collection of regression basis functions is closely related to the problem of active input identification. Again, this is a model selection problem where we now have a reduced set of possible basis functions, all of which now only involve the active inputs. We apply the same methods as above, only this time removing or adding single regression terms in order to arrive at an appropriate and parsimonious representation for the simulator output.

Estimate emulator regression coefficients

The next stage is to quantify beliefs about the emulator coefficients β. We obtain values for these prior quantities by fitting the mean function that we have determined in the previous stages to the observed simulator runs to obtain estimates and associated uncertainty statements about the coefficients β.

There are a variety of ways in which we could fit the mean function to obtain the estimates $\hat{\beta}$. If we lack any insight into the nature of the correlation structure of the residual process and our design points are well-separated then we could fit the regression model using ordinary least squares (OLS). Alternatively, if we have information about the correlation function and its parameter values then more appropriate estimates could be obtained by using this information and fitting by generalised least squares (GLS). We might use an iterative approach to estimate both the regression coefficients and the correlation function hyperparameters.

The value of $\mathrm{E}[\beta]$ is then taken to be the estimate $\hat{\beta}$ from the fitting of the regression model and $\mathrm{Var}[\beta]$ is taken to be the variance of the corresponding estimates. With sufficient evaluations in an approximately orthogonal design, the estimation error here is negligible.

8.32.5 Estimate residual process hyperparameters

The final stage is to make assessments for the values of the covariance function hyperparameters (σ^2, δ) in our covariance specifications for the residual process $w(x)$.

Typically an estimate for σ^2 is obtained from fitting the regression model in the form of the residual mean square $\hat{\sigma}^2$. Estimating correlation function hyperparameters for the emulator residuals can be a more complex task, which is discussed in the alternatives page on estimators of correlation hyperparameters (*AltEstimateDelta*). A common empirical approach is *variogram fitting*.

Validation and post-emulation tasks

Given the emulator, we can perform similar diagnostics, validation, and post-emulation tasks as described in the thread for Bayes linear emulation for the core model (*ThreadCoreBL*).

References

Craig, P. S., Goldstein, M., Seheult, A. H., and Smith, J. A. (1998), "Constructing partial prior specifications for models of complex physical systems," *Applied Statistics*, 47, 37-53.

8.33 Procedure: Build Gaussian process emulator for the core problem

8.33.1 Description and Background

The preparation for building a *Gaussian process* (GP) *emulator* for the *core problem* involves defining the prior mean and covariance functions, identifying prior distributions for *hyperparameters*, creating a *design* for the *training sample*, then running the *simulator* at the input configurations specified in the design. All of this is described in the thread for the analysis of the core model using Gaussian process methods (*ThreadCoreGP*). The procedure here is for taking those various ingredients and creating the GP emulator.

8.33.2 Inputs

- GP prior mean function $m(\cdot)$ depending on hyperparameters β
- GP prior correlation function $c(\cdot, \cdot)$ depending on hyperparameters δ
- Prior distribution $\pi(\cdot, \cdot, \cdot)$ for β, σ^2 and δ, where σ^2 is the process variance hyperparameter
- Design D comprising points $\{x_1, x_2, \ldots, x_n\}$ in the input space
- Output vector $f(D) = (f(x_1), f(x_2), \ldots, f(x_n))^T$, where $f(x_j)$ is the simulator output from input point x_j

8.33.3 Outputs

A GP-based emulator in one of the forms presented in the discussion page on GP emulator forms (*DiscGPBasedEmulator*).

In the case of general prior mean and correlation functions and general prior distribution:

- A GP posterior conditional distribution with mean function $m^*(\cdot)$ and covariance function $v^*(\cdot, \cdot)$ conditional on $\theta = \{\beta, \sigma^2, \delta\}$

- A posterior representation for θ

In the case of linear mean function, general correlation function, weak prior information on β, σ^2 and general prior distribution for δ:

- A *t process* posterior conditional distribution with mean function $m^*(\cdot)$, covariance function $v^*(\cdot, \cdot)$ and degrees of freedom b^* conditional on δ
- A posterior representation for δ

As explained in *DiscGPBasedEmulator*, the "posterior representation" for the hyperparameters is formally the posterior distribution for those hyperparameters, but for computational purposes this distribution is represented by a sample of hyperparameter values. In either case, the outputs define the emulator and allow all necessary computations for tasks such as prediction of the simulator output, *uncertainty analysis* or *sensitivity analysis*.

8.33.4 Procedure

General case

Define the following arrays, according to the conventions set out in the notation page (*MetaNotation*).

$e = f(D) - m(D)$, an $n \times 1$ vector;

$A = c(D, D)$, an $n \times n$ matrix;

$t(x) = c(D, x)$, an $n \times 1$ vector function of x.

Then, conditional on θ and the training sample, the simulator output $f(x)$ is a GP with posterior mean function

$$m^*(x) = m(x) + t(x)^{\mathrm{T}} A^{-1} e$$

and posterior covariance function

$$v^*(x, x') = \sigma^2 \{c(x, x') - t(x)^{\mathrm{T}} A^{-1} t(x')\}.$$

This is the first part of the emulator as discussed in *DiscGPBasedEmulator*. The emulator is completed by a second part formally comprising the posterior distribution of θ, which has density given by

$$\pi^*(\beta, \sigma^2, \delta) \propto \pi(\beta, \sigma^2, \delta) \times (\sigma^2)^{-n/2} |A|^{-1/2} \times \exp\{-e^{\mathrm{T}} A^{-1} e/(2\sigma^2)\},$$

where the symbol $\propto$ denotes proportionality as usual in Bayesian statistics. In order to compute the emulator predictions and other tasks, the posterior representation of θ includes a sample from this posterior distribution. The standard method for obtaining this is Markov chain Monte Carlo (MCMC). For this general case, the form of the posterior distribution depends very much on the forms of prior mean and correlation functions and the prior distribution, so no general advice can be given. The References section below lists some useful texts on MCMC.

Linear mean and weak prior case

Suppose now that the mean function has the linear form $m(x) = h(x)^{\mathrm{T}} \beta$, where $h(\cdot)$ is a vector of q known *basis functions* of the inputs and β is a $q \times 1$ column vector of hyperparameters. Suppose also that the prior distribution has the form $\pi(\beta, \sigma^2, \delta) \propto \sigma^{-2} \pi_\delta(\delta)$, i.e. that we have weak prior information on β and σ^2 and an arbitrary prior distribution $\pi_\delta(\cdot)$ for δ.

Define A and $t(\cdot)$ as in the previous case. In addition, define the $n \times q$ matrix

$$H = [h(x_1), h(x_2), \ldots, h(x_n)]^{\mathrm{T}},$$

or in a more compact notation as $H = h(D^{\mathrm{T}})^{\mathrm{T}}$, the vector

$$\widehat{\beta} = \left(H^{\mathrm{T}} A^{-1} H\right)^{-1} H^{\mathrm{T}} A^{-1} f(D),$$

and the scalar

$$\widehat{\sigma}^2 = (n-q-2)^{-1} f(D)^{\mathrm{T}} \left\{ A^{-1} - A^{-1} H \left(H^{\mathrm{T}} A^{-1} H\right)^{-1} H^{\mathrm{T}} A^{-1} \right\} f(D),$$

which can also be written as

$$\widehat{\sigma}^2 = (n-q-2)^{-1} (f(D) - H\hat{\beta})^{\mathrm{T}} A^{-1} (f(D) - H\hat{\beta}).$$

Then, conditional on δ and the training sample, the simulator output $f(x)$ is a t process with $b^* = n - q$ degrees of freedom, posterior mean function

$$m^*(x) = h(x)^{\mathrm{T}} \widehat{\beta} + t(x)^{\mathrm{T}} A^{-1} (f(D) - H\widehat{\beta})$$

and posterior covariance function

$$v^*(x, x') = \widehat{\sigma}^2 \{ c(x, x') - t(x)^{\mathrm{T}} A^{-1} t(x') + \left(h(x)^{\mathrm{T}} - t(x)^{\mathrm{T}} A^{-1} H\right) \left(H^{\mathrm{T}} A^{-1} H\right)^{-1} \left(h(x')^{\mathrm{T}} - t(x')^{\mathrm{T}} A^{-1} H\right)^{\mathrm{T}} \}.$$

This is the first part of the emulator as discussed in *DiscGPBasedEmulator*. The emulator is formally completed by a second part comprising the posterior distribution of δ, which has density given by

$$\pi^*_\delta(\delta) \propto \pi_\delta(\delta) \times (\widehat{\sigma}^2)^{-(n-q)/2} |A|^{-1/2} |H^{\mathrm{T}} A^{-1} H|^{-1/2}.$$

In order to derive the sample representation of this posterior distribution for the second part of the emulator, three approaches can be considered.

1. A common approximation is simply to fix δ at a single value estimated from the posterior distribution. The usual choice is the posterior mode, which can be found as the value of δ for which $\pi^*(\delta)$ is maximised. The discussion page on finding the posterior mode of delta (*DiscPostModeDelta*), presents some details of this procedure. See also the alternatives page on estimators of correlation hyperparameters (*AltEstimateDelta*) for a discussion of alternative estimators.

2. Another approach is to formally account for the uncertainty about the true value of δ, by sampling the posterior distribution of the correlation lengths and performing a Monte Carlo integration. This is described in the procedure page *ProcMCMCDeltaCoreGP*. A reference on MCMC algorithms can be found below.

3. An intermediate approach first approximates the posterior distribution by a multivariate lognormal distribution and then uses a sample from this distribution. See also the procedure on multivariate lognormal approximation for correlation hyperparameters (*ProcApproxDeltaPosterior*).

Each of these approaches results in a set of values (or just a single value in the case of the first approach) of δ, which allow the emulator predictions and other required inferences to be computed.

Although it represents an approximation that ignores the uncertainty in δ, approach 1 has been widely used. It has often been suggested that, although uncertainty in these correlation hyperparameters can be substantial, taking proper account of that uncertainty through approach 2 does not lead to appreciable differences in the resulting emulator. On the other hand, although this may be true if a good single estimate for δ is used, this is not necessarily easy to find, and the posterior mode may sometimes be a poor choice. Approach 3 has not been used much, but can be recommended when there is concern about using just a single δ estimate. It is simpler than the full MCMC approach 2, but should capture the uncertainty in δ well.

Approaches 1 and 2 are both used in the GEM-SA software (*disclaimer*).

8.33.5 Additional Comments

Several computational issues can arise in implementing this procedure. These are discussed in *DiscBuildCoreGP*.

8.33.6 References

Here are two leading textbooks on MCMC:

- Gilks, W.R., Richardson, S. & Spiegelhalter, D.J. (1996). Markov Chain Monte Carlo in Practice. Chapman & Hall.
- Gamerman, D. and Lopes, H. F. (2006). Markov Chain Monte Carlo: Stochastic Simulation for Bayesian Inference. CRC Press.

Although MCMC for the distribution of δ has been reported in a number of articles, they have not given any details for how to do this, assuming instead that the reader is familiar with MCMC techniques.

8.34 Procedure: Build Gaussian process emulator of derivatives

8.34.1 Description and Background

The preparation for building a *Gaussian process* (GP) *emulator* of derivatives involves defining the prior mean and covariance functions, identifying prior distributions for *hyperparameters*, creating a *design* for the *training sample*, then running the *adjoint*, or *simulator*, at the input configurations specified in the design. This is described in the generic thread on methods to emulate derivatives (*ThreadGenericEmulateDerivatives*). The procedure here is for taking those various ingredients and creating the GP emulator.

8.34.2 Additional notation for this page

Derivative information requires further notation than is specified in the Toolkit notation page (*MetaNotation*). As in the procedure page on building a GP emulator with derivative information (*ProcBuildWithDerivsGP*) we use the following additional notation:

- The tilde symbol () placed over a letter denotes derivative information and function output combined.
- We introduce an extra argument to denote a derivative. We define $\tilde{f}(x,d)$ to be the derivative of $f(x)$ with respect to input d and so $d \in \{0,1,\ldots,p\}$. When $d=0$ we have $\tilde{f}(x,0)=f(x)$. For simplicity, when $d=0$ we adopt the shorter notation so we use $f(x)$ rather than $\tilde{f}(x,0)$.
- An input is denoted by a superscript on x, while a subscript on x refers to the point in the input space. For example, $x_i^{(k)}$ refers to input k at point i.

8.34.3 Inputs

- GP prior mean function $m(\cdot)$, differentiable and depending on hyperparameters β
- GP prior correlation function $c(\cdot,\cdot)$, twice differentiable and depending on hyperparameters δ
- Prior distribution $\pi(\cdot,\cdot,\cdot)$ for β,σ^2 and δ where Σ is the process variance hyperparameter
- Design, $\tilde{D}=\{(x_k,d_k)\}$, where $k=\{1,\ldots,\tilde{n}\}$ and $d_k \in \{0,1,\ldots,p\}$. We have x_k which refers to the location in the design and d_k determines whether at point x_k we require function output or a first derivative w.r.t one of the inputs. Each x_k is not necessarily distinct as we may have a derivative and the function output at point x_k or we may require a derivative w.r.t several inputs at point x_k. If we do not have any derivative information, $d_k=0,\forall k$ and the resulting design is as in the core thread *ThreadCoreGP*.
- Output vector is $\tilde{f}(\tilde{D})=\tilde{f}(x_k,d_k)$ of length $\tilde{n}$. If we are not including derivatives in the training data, $d_k=0,\forall k$ and the output vector reduces to $f(D)=f(x)$ as in *ThreadCoreGP*.

8.34.4 Outputs

A GP-based emulator in one of the forms discussed in the discussion page *DiscGPBasedEmulator*.

In the case of general prior mean and correlation functions and general prior distribution:

- A GP posterior conditional distribution with mean function $\tilde{m}^*(\cdot)$ and covariance function $\tilde{v}^*(\cdot,\cdot)$ conditional on $\theta = \{\beta, \sigma^2, \delta\}$.
- A posterior representation for θ

In the case of linear mean function, general correlation function, weak prior information on β, σ^2 and general prior distribution for δ:

- A *t process* posterior conditional distribution with mean function $\tilde{m}^*(\cdot)$, covariance function $\tilde{v}^*(\cdot,\cdot)$ and degrees of freedom b^* conditional on δ
- A posterior representation for δ

As explained in *DiscGPBasedEmulator*, the "posterior representation" for the hyperparameters is formally the posterior distribution for those hyperparameters, but for computational purposes this distribution is represented by a sample of hyperparameter values. In either case, the outputs define the emulator and allow all necessary computations for tasks such as prediction of the partial derivatives of the simulator output w.r.t the inputs, *uncertainty analysis* or *sensitivity analysis*.

8.34.5 Procedure

General case

We define the following arrays (following the conventions set out in *MetaNotation* where possible).

$\tilde{e} = \tilde{f}(\tilde{D}) - \tilde{m}(\tilde{D})$, an $\tilde{n} \times 1$ vector, where $\tilde{m}(\tilde{D}) = \frac{\partial}{\partial x^{(d_k)}} m(x_k)$.

$\tilde{A} = \tilde{c}(\tilde{D}, \tilde{D})$, an $\tilde{n} \times \tilde{n}$ matrix, where $\tilde{c}(\cdot,\cdot)$ includes the covariances involving derivatives. The exact form of $\tilde{c}(\cdot,\cdot)$ depends on where derivatives are included. The general expression for this is: $\tilde{c}(\cdot,\cdot) = \mathrm{Corr}\{\tilde{f}(x_i, d_i), \tilde{f}(x_j, d_j)\}$ and we can break it down into three cases:

- Case 1 is for when $d_i = d_j = 0$ and as such represents the covariance between 2 points. This is the same as in *ThreadCoreGP* and is given by:

$$\mathrm{Corr}\{\tilde{f}(x_i, 0), \tilde{f}(x_j, 0)\} = c(x_i, x_j).$$

- Case 2 is for when $d_i \neq 0$ and $d_j = 0$ and as such represents the covariance between a derivative and a point. This is obtained by differentiating $c(\cdot,\cdot)$ w.r.t input d_i:

$$\mathrm{Corr}\{\tilde{f}(x_i, d_i), \tilde{f}(x_j, 0)\} = \frac{\partial c(x_i, x_j)}{\partial x_i^{(d_i)}}, \text{for } d_i \neq 0.$$

- Case 3 is for when $d_i \neq 0$ and $d_j \neq 0$ and as such represents the covariance between two derivatives. This is obtained by differentiating $c(\cdot,\cdot)$ twice: once w.r.t input d_i and once w.r.t input d_j:

$$\mathrm{Corr}\{\tilde{f}(x_i, d_j), \tilde{f}(x_j, d_j)\} = \frac{\partial^2 c(x_i, x_j)}{\partial x_i^{(d_i)} \partial x_j^{(d_j)}}, \text{for } d_i, d_j \neq 0.$$

  - Case 3a. If $d_i, d_j \neq 0$ and $d_i = d_j$ we have a special version of Case 3 which gives:

$$\mathrm{Corr}\{\tilde{f}(x_i, d_i), \tilde{f}(x_j, d_i)\} = \frac{\partial^2 c(x_i, x_j)}{\partial x_i^{(d_i)}, x_j^{(d_i)}}, \text{for } d_i \neq 0.$$

$\tilde{t}(x,d) = \tilde{c}\{\tilde{D},(x,d)\}$, an $\tilde{n}\times 1$ vector function of x. We have $d \neq 0$ as here we want to emulate derivatives. To emulate function output, $d = 0$ and this is covered in *ThreadCoreGP* or *ThreadVariantWithDerivatives* if we have derivatives in the training data.

Then, conditional on θ and the training sample, the output vector $\tilde{f}(x,d)$ is a multivariate GP with posterior mean function

$$\tilde{m}^*(x,d) = \tilde{m}(x,d) + \tilde{t}(x,d)^{\mathrm{T}}\tilde{A}^{-1}\tilde{e}$$

and posterior covariance function

$$\tilde{v}^*\{(x_i,d_i),(x_j,d_j)\} = \sigma^2\{\tilde{c}\{(x_i,d_i),(x_j,d_j)\} - \tilde{t}(x_i,d_i)^{\mathrm{T}}\tilde{A}^{-1}\tilde{t}(x_j,d_j)\}.$$

This is the first part of the emulator as discussed in *DiscGPBasedEmulator*. The emulator is completed by a second part formally comprising the posterior distribution of θ, which has density given by

$$\pi^*(\beta,\sigma^2,\delta) \propto \pi(\beta,\sigma^2,\delta)\times(\sigma^2)^{-\tilde{n}/2}|\tilde{A}|^{-1/2}\times\exp\{-\tilde{e}^{\mathrm{T}}\tilde{A}^{-1}\tilde{e}/(2\sigma^2)\}.$$

For the output vector $\tilde{f}(x,0) = f(x)$ see the procedure page on building a GP emulator for the core problem (*ProcBuildCoreGP*) or the procedure page for building a GP emulator when we have derivatives in the training data (*ProcBuildWithDerivsGP*).

Linear mean and weak prior case

Suppose now that the mean function has the linear form $m(x) = h(x)^{\mathrm{T}}\beta$, where $h(\cdot)$ is a vector of q known *basis functions* of the inputs and β is a $q\times 1$ column vector of hyperparameters. When $d\neq 0$ we therefore have $\tilde{m}(x,d) = \tilde{h}(x,d)^{\mathrm{T}}\beta = \frac{\partial}{\partial x^{(d)}}h(x)^{\mathrm{T}}\beta$. Suppose also that the prior distribution has the form $\pi(\beta,\Sigma,\delta)\propto\sigma^{-2}\pi_\delta(\delta)$, i.e. that we have weak prior information on β and Σ and an arbitrary prior distribution $\pi_\delta(\cdot)$ for δ.

Define $\tilde{A}$ and $\tilde{t}(x)$ as in the previous case. In addition, define the $\tilde{n}\times q$ matrix

$$\tilde{H} = [\tilde{h}(x_1,d_1),\dots,\tilde{h}(x_{\tilde{n}},d_{\tilde{n}})]^{\mathrm{T}},$$

the vector

$$\widehat{\beta} = \left(\tilde{H}^{\mathrm{T}}\tilde{A}^{-1}\tilde{H}\right)^{-1}\tilde{H}^{\mathrm{T}}\tilde{A}^{-1}\tilde{f}(\tilde{D})$$

and the scalar

$$\widehat{\sigma}^2 = (\tilde{n}-q-2)^{-1}\tilde{f}(\tilde{D})^{\mathrm{T}}\left\{\tilde{A}^{-1} - \tilde{A}^{-1}\tilde{H}\left(\tilde{H}^{\mathrm{T}}\tilde{A}^{-1}\tilde{H}\right)^{-1}\tilde{H}^{\mathrm{T}}\tilde{A}^{-1}\right\}\tilde{f}(\tilde{D}).$$

Then, conditional on δ and the training sample, the output vector $\tilde{f}(x,d)$ is a t process with $b^* = \tilde{n}-q$ degrees of freedom, posterior mean function

$$\tilde{m}^*(x,d) = \tilde{h}(x,d)^{\mathrm{T}}\widehat{\beta} + \tilde{t}(x,d)^{\mathrm{T}}\tilde{A}^{-1}(\tilde{f}(\tilde{D}) - \tilde{H}\widehat{\beta})$$

and posterior covariance function

$$\begin{aligned}\tilde{v}^*\{(x_i,d_i),(x_j,d_j)\} =& \widehat{\sigma}^2\{\tilde{c}\{(x_i,d_i),(x_j,d_j)\} - \tilde{t}(x_i,d_i)^{\mathrm{T}}\tilde{A}^{-1}\tilde{t}(x_j,d_j)\\ &+ \left(\tilde{h}(x_i,d_i)^{\mathrm{T}} - \tilde{t}(x_i,d_i)^{\mathrm{T}}\tilde{A}^{-1}\tilde{H}\right)\left(\tilde{H}^{\mathrm{T}}\tilde{A}^{-1}\tilde{H}\right)^{-1}\left(\tilde{h}(x_j,d_j)^{\mathrm{T}} - \tilde{t}(x_j,d_j)^{\mathrm{T}}\tilde{A}^{-1}\tilde{H}\right)^{\mathrm{T}}\}.\end{aligned}$$

This is the first part of the emulator as discussed in *DiscGPBasedEmulator*. The emulator is formally completed by a second part comprising the posterior distribution of δ, which has density given by

$$\pi^*_\delta(\delta)\propto\pi_\delta(\delta)\times(\widehat{\sigma}^2)^{-(\tilde{n}-q)/2}|\tilde{A}|^{-1/2}|\tilde{H}^{\mathrm{T}}\tilde{A}^{-1}\tilde{H}|^{-1/2}.$$

In order to derive the sample representation of this posterior distribution for the second part of the emulator, three approaches can be considered.

1. Exact computations require a sample from the posterior distribution of δ. This can be obtained by MCMC; a suitable reference can be found below.
2. A common approximation is simply to fix δ at a single value estimated from the posterior distribution. The usual choice is the posterior mode, which can be found as the value of δ for which $\pi^*(\delta)$ is maximised. See the alternatives page *AltEstimateDelta* for a discussion of alternative estimators.
3. An intermediate approach first approximates the posterior distribution by a multivariate lognormal distribution and then uses a sample from this distribution; this is described in the procedure page *ProcApproxDeltaPosterior*.

Each of these approaches results in a set of values (or just a single value in the case of the second approach) of δ, which allow the emulator predictions and other required inferences to be computed.

Although it represents an approximation that ignores the uncertainty in δ, approach 2 has been widely used. It has often been suggested that, although uncertainty in these correlation hyperparameters can be substantial, taking proper account of that uncertainty through approach 1 does not lead to appreciable differences in the resulting emulator. On the other hand, although this may be true if a good single estimate for δ is used, this is not necessarily easy to find, and the posterior mode may sometimes be a poor choice. Approach 3 has not been used much, but can be recommended when there is concern about using just a single δ estimate. It is simpler than the full MCMC approach 1, but should capture the uncertainty in δ well.

8.34.6 Additional Comments

We can use this procedure to emulate derivatives whether or not we have derivatives in the training data. Quantities $\tilde{A}, \tilde{H}, \tilde{f}(\tilde{D}), \tilde{m}(\tilde{D})$ and therefore $\tilde{e}$, above are taken from *ProcBuildWithDerivsGP* as they allow for derivatives in the training data, in addition to function output. In the case when we build an emulator with function output only, $d = 0$ for all the training data and these quantities reduce to the same quantities without the tilde symbol (), as defined in *ProcBuildCoreGP*. Then to emulate derivatives in the general case, conditional on θ and the training sample, the output vector $\tilde{f}(x,d)$ is a multivariate GP with posterior mean function

$$\tilde{m}^*(x,d) = \tilde{m}(x,d) + \tilde{t}(x,d)^{\mathrm{T}} A^{-1} e$$

and posterior covariance function

$$\tilde{v}^*\{(x_i,d_i),(x_j,d_j)\} = \sigma^2\{\tilde{c}\{(x_i,d_i),(x_j,d_j)\} - \tilde{t}(x_i,d_i)^{\mathrm{T}} A^{-1} \tilde{t}(x_j,d_j)\}.$$

To emulate derivatives in the case of a linear mean and weak prior, conditional on δ and the training sample, the output vector $\tilde{f}(x,d)$ is a t process with $b^* = n - q$ degrees of freedom, posterior mean function

$$\tilde{m}^*(x,d) = \tilde{h}(x,d)^{\mathrm{T}}\widehat{\beta} + \tilde{t}(x,d)^{\mathrm{T}} A^{-1}(f(D) - H\widehat{\beta})$$

and posterior covariance function

$$\begin{aligned}\tilde{v}^*\{(x_i,d_i),(x_j,d_j)\} =& \widehat{\sigma}^2\{\tilde{c}\{(x_i,d_i),(x_j,d_j)\} - \tilde{t}(x_i,d_i)^{\mathrm{T}} A^{-1}\tilde{t}(x_j,d_j) \\ &+ \left(\tilde{h}(x_i,d_i)^{\mathrm{T}} - \tilde{t}(x_i,d_i)^{\mathrm{T}} A^{-1} H\right)\left(H^{\mathrm{T}} A^{-1} H\right)^{-1}\left(\tilde{h}(x_j,d_j)^{\mathrm{T}} - \tilde{t}(x_j,d_j)^{\mathrm{T}} A^{-1} H\right)^{\mathrm{T}}\}.\end{aligned}$$

8.35 Procedure: Build multivariate Gaussian process emulator for the core problem

8.35.1 Description and Background

The preparation for building a *multivariate Gaussian process* (GP) *emulator* for the *core problem* involves defining the prior mean and covariance functions, identifying prior distributions for *hyperparameters*, creating a *design* for the *training sample*, then running the *simulator* at the input configurations specified in the design. All of this is described in the variant thread for analysis of a simulator with multiple outputs using Gaussian Process methods (*ThreadVariantMultipleOutputs*). The procedure here is for taking those various ingredients and creating the GP emulator.

In the case of r outputs, the simulator is a $1 \times r$ row vector function $f(\cdot)$,and so its mean function $m(\cdot)$ is also a $1 \times r$ row vector function, while its covariance function $v(\cdot, \cdot)$ is a $r \times r$ matrix function. The i, jth element of $v(\cdot, \cdot)$ expresses the covariance between $f_i(\cdot)$ and $f_j(\cdot)$.

8.35.2 Inputs

- GP prior mean function $m(\cdot)$ depending on hyperparameters β
- GP prior input-space covariance function $v(\cdot, \cdot)$ depending on hyperparameters ω
- Prior distribution $\pi(\cdot, \cdot)$ for β and ω.
- Design D comprising points $\{x_1, x_2, \ldots, x_n\}$ in the input space.
- $n \times r$ output matrix $f(D)$.

8.35.3 Outputs

A GP-based emulator in one of the forms presented in the discussion page *DiscGPBasedEmulator*.

In the case of general prior mean and correlation functions and general prior distribution

- A multivariate GP posterior conditional distribution with mean function $m^*(\cdot)$ and covariance function $v^*(\cdot, \cdot)$ conditional on $\theta = \{\beta, \omega\}$.
- A posterior representation for θ.

In the case of linear mean function, general covariance function, weak prior information on β and general prior distribution for ω we have:

- A *multivariate Gaussian process* posterior conditional distribution with mean function $m^*(\cdot)$ and covariance function $v^*(\cdot, \cdot)$ conditional on ω.
- A posterior representation for ω.

As explained in *DiscGPBasedEmulator*, the "posterior representation" for the hyperparameters is formally the posterior distribution for those hyperparameters, but for computational purposes this distribution is represented by a sample of hyperparameter values. In either case, the outputs define the emulator and allow all necessary computations for tasks such as prediction of the simulator output, *uncertainty analysis* or *sensitivity analysis*.

8.35.4 Procedure

General case

We define the following arrays (following the conventions set out in the Toolkit's notation page (*MetaNotation*)).

- $e = f(D) - m(D)$, an $n \times r$ matrix;
- $V = v(D, D)$, the $rn \times rn$ covariance matrix composed of $n \times n$ blocks $\{V_{ij} : i, j = 1, ..., r\}$, where the k, ℓth entry of V_{ij} is the covariance between $f_i(x_k)$ and $f_j(x_\ell)$;
- $u(x) = v(D, x)$, the $rn \times r$ matrix function of x composed of $n \times 1$ blocks $\{u_{ij}(x) : i, j = 1, ..., r\}$, where the k is the covariance between $f_i(x_k)$ and $f_j(x)$.

Then, conditional on θ and the training sample, the simulator output vector $f(x)$ is a multivariate GP with posterior mean function

$$m^*(x) = m(x) + \mathrm{vec}(e)^{\mathrm{T}} V^{-1} u(x)$$

and posterior covariance function

$$v^*(x,x') = v(x,x') - u(x)^{\mathrm{T}} V^{-1} u(x').$$

This is the first part of the emulator as discussed in *DiscGPBasedEmulator*. The emulator is completed by a second part formally comprising the posterior distribution of θ, which has density given by

$$\pi^*(\beta,\omega) \propto \pi(\beta,\omega) \times |V|^{-1/2} \exp\left\{-\frac{1}{2}\mathrm{vec}(e)^{\mathrm{T}} V^{-1} \mathrm{vec}(e)\right\}$$

where the symbol $\propto$ denotes proportionality as usual in Bayesian statistics. In order to compute the emulator predictions and other tasks, the posterior representation of θ includes a sample from this posterior distribution. The standard method for doing this is Markov chain Monte Carlo (MCMC). For this general case, the form of the posterior distribution depends very much on the forms of prior mean and correlation functions and the prior distribution, so no general advice can be given. The References section below lists some useful texts on MCMC.

Linear mean and weak prior case

Suppose now that the mean function has the linear form $m(x) = h(x)^{\mathrm{T}}\beta$, where $h(\cdot)$ is a vector of q known *basis functions* of the inputs and β is a $q \times r$ matrix of hyperparameters. Suppose also that the prior distribution has the form $\pi(\beta,\omega) \propto \pi_\omega(\omega)$, i.e. that we have weak prior information on β and an arbitrary prior distribution $\pi_\omega(\cdot)$ for ω.

Define V and $u(x)$ as in the previous case. In addition, define the $n \times q$ matrix

$$H = h(D)^{\mathrm{T}},$$

the $q \times r$ matrix $\widehat{\beta} =$ such that

$$\mathrm{vec}(\widehat{\beta}) = \left((I_k \otimes H^{\mathrm{T}})V^{-1}(I_k \otimes H)\right)^{-1} (I_k \otimes H^{\mathrm{T}})V^{-1}\mathrm{vec}(f(D)),$$

and the $r \times qr$ matrix

$$R(x) = I_k \otimes h(x)^{\mathrm{T}} - u(x)^{\mathrm{T}} V^{-1}(I_k \otimes H).$$

Then, conditional on ω and the training sample, the simulator output vector $f(x)$ is a *multivariate GP* with posterior mean function

$$m^*(x) = h(x)^T\widehat{\beta} + u(x)^{\mathrm{T}} V^{-1}\mathrm{vec}(f(D) - H\widehat{\beta})$$

and posterior covariance function

$$v^*(x,x') = v(x,x') - u(x)^{\mathrm{T}} V^{-1} u(x') + R(x)\left((I_k \otimes H^{\mathrm{T}})V^{-1}(I_k \otimes H)\right)^{-1} R(x')^{\mathrm{T}}.$$

This is the first part of the emulator as discussed in *DiscGPBasedEmulator*. The emulator is formally completed by a second part comprising the posterior distribution of ω, which has density given by

$$pi^*_\omega(\omega) \propto \pi_\omega(\omega) \times |V|^{-1/2} \|(I_k \otimes H^{\mathrm{T}})V^{-1}(I_k \otimes H)|^{-1/2} \exp\left\{-\frac{1}{2}\mathrm{vec}(f(D) - H\widehat{\beta})^{\mathrm{T}} V^{-1}\mathrm{vec}(f(D) - H\widehat{\beta})\right\}.$$

In order to compute the emulator predictions and other tasks, the posterior representation of θ includes a sample from this posterior distribution. The standard method for doing this is Markov chain Monte Carlo (MCMC). For this general case, the form of the posterior distribution depends very much on the forms of prior mean and correlation functions and the prior distribution, so no general advice can be given. The References section below lists some useful texts on MCMC.

Choice of covariance function

The procedures above are for a general multivariate covariance function $v(\cdot,\cdot)$. As such, the emulators are conditional on the choice of covariance function $v(\cdot,\cdot)$ and its associated hyperparameters ω. In order to use the emulator, a structure for $v(\cdot,\cdot)$ must be chosen that ensures the covariance matrix $v(D,D)$ is positive semi-definite for any design D. The options for this structure are found in the alternatives page *AltMultivariateCovarianceStructures*.

The simplest option is the *separable* structure. In many cases the separable structure is adequate, and leads to several simplifications in the above mathematics. The result is an easily built and workable multi-output emulator. The aforementioned mathematical simplifications, and the procedure for completing the separable multi-output emulator, are in the procedure page *ProcBuildMultiOutputGPSep*.

More complex, nonseparable structures are available and can provide greater flexibility than the separable structure, but at the cost of producing emulators that are harder to build. Options for nonseparable covariance functions are discussed in *AltMultivariateCovarianceStructures*.

8.35.5 Additional Comments

Several computational issues can arise in implementing this procedure. These are discussed in *DiscBuildCoreGP*.

8.35.6 References

Here are two leading textbooks on MCMC:

- Gilks, W.R., Richardson, S. & Spiegelhalter, D.J. (1996). Markov Chain Monte Carlo in Practice. Chapman & Hall.
- Gamerman, D. and Lopes, H. F. (2006). Markov Chain Monte Carlo: Stochastic Simulation for Bayesian Inference. CRC Press.

Although MCMC for the distribution of δ has been reported in a number of articles, they have not given any details for how to do this, assuming instead that the reader is familiar with MCMC techniques.

8.36 Procedure: Complete the multivariate Gaussian process emulator with a separable covariance function

8.36.1 Description and Background

The first steps in building a multivariate Gaussian process emulator for a simulator with r outputs are described in *ProcBuildMultiOutputGP*. We assume here that a linear mean function $m(x) = h(x)^{\mathrm{T}}\beta$ with a weak prior $\pi(\beta) \propto 1$ is used in that procedure, so the result is a multivariate Gaussian process emulator that is conditional on the choice of covariance function $v(\cdot,\cdot)$ and its associated hyperparameters ω. It is necessary to choose one of the structures for $v(\cdot,\cdot)$ discussed in *AltMultivariateCovarianceStructures*. The most simple option is a *separable* structure which leads to a simpler multivariate emulator than that described in *ProcBuildMultiOutputGP*. The procedure here is for creating that simplified multivariate Gaussian process emulator with a *separable* covariance function.

A separable covariance has the form

$$v(\cdot,\cdot) = \Sigma c(\cdot,\cdot),$$

where Σ is a :math:‘ r times r‘ covariance matrix between outputs and $c(\cdot,\cdot)$ is a correlation function between input points. The hyperparameters for the separable covariance are $\omega = (\Sigma, \delta)$, where δ are the hyperparameters for $c(\cdot,\cdot)$.

The choice of prior for Σ is discussed in *AltMultivariateGPPriors*, but here we assume here that Σ has the weak prior $\pi_\Sigma(\Sigma) \propto |\Sigma|^{-\frac{r+1}{2}}$.

As discussed in *AltMultivariateCovarianceStructures*, the assumption of separability imposes a restriction that all the outputs have the same correlation function $c(\cdot,\cdot)$ across the input space. We shall see in the following procedure that this leads to a simple emulation methodology. The drawback is that the emulator may not perform well if the outputs represent several different types of physical quantity, since it assumes that all outputs have the same smoothness properties. If that assumption is too restrictive, then a nonseparable covariance function may be required. Some options for nonseparable covariance function are described in *AltMultivariateCovarianceStructures*.

8.36.2 Inputs

- Multivariate GP emulator with a linear mean function that is conditional on the choice of covariance function $v(\cdot,\cdot)$ and its associated hyperparameters ω, which is constructed according to the procedure in *ProcBuildMultiOutputGP*.
- A GP prior input-space correlation function $c(\cdot,\cdot)$ depending on hyperparameters δ
- A prior distribution $\pi(\cdot)$ for δ.

8.36.3 Outputs

- A GP-based emulator with a *multivariate t-process* posterior conditional distribution with mean function $m^*(\cdot)$, covariance function $v^*(\cdot,\cdot)$ and degrees of freedom b^* conditional on δ.
- A posterior representation for δ

As explained in *DiscGPBasedEmulator*, the "posterior representation" for the hyperparameters is formally the posterior distribution for those hyperparameters, but for computational purposes this distribution is represented by a sample of hyperparameter values. In either case, the outputs define the emulator and allow all necessary computations for tasks such as prediction of the simulator output, *uncertainty analysis* or *sensitivity analysis*.

8.36.4 Procedure

In addition to the notation defined in *ProcBuildMultiOutputGP*, we define the following arrays (following the conventions set out in the Toolkit's notation page *MetaNotation*).

- $A = c(D,D)$, the $n \times n$ matrix of input-space correlations between all pairs of design points in D;
- $t(x) = c(D,x)$, an $n \times 1$ vector function of x.
- $R(x) = h(x)^{\mathrm{T}} - t(x)^{\mathrm{T}} A^{-1} H$

A consequence of the separable structure for $v(\cdot,\cdot)$ is that the $rn \times rn$ covariance matrix $V = v(D,D)$ has the Kronecker product representation $V = \Sigma \otimes A$, and the $rn \times r$ matrix function $u(x) = v(D,x)$ has the Kronecker product representation $u(x) = \Sigma \otimes t(x)$. As a result the $n \times r$ matrix $\widehat{\beta}$ has the simpler form

$$\widehat{\beta} = \left(H^{\mathrm{T}} A^{-1} H\right)^{-1} H^{\mathrm{T}} A^{-1} f(D).$$

Then, conditional on δ and the training sample, the simulator output vector $f(x)$ is a *multivariate t-process* with $b^* = n - q$ degrees of freedom, posterior mean function

$$m^*(x) = h(x)^T \widehat{\beta} + t(x)^{\mathrm{T}} A^{-1}(f(D) - H\widehat{\beta})$$

and posterior covariance function

$$v^*(x,x') = \widehat{\Sigma}\left\{c(x,x') - t(x)^{\mathrm{T}} A^{-1} t(x') + R(x)\left(H^{\mathrm{T}} A^{-1} H\right)^{-1} R(x')^{\mathrm{T}}\right\},$$

where

$$\begin{aligned}\widehat{\Sigma} &= (n-q)^{-1}(f(D)-H\widehat{\beta})^{\mathrm{T}}A^{-1}(f(D)-H\widehat{\beta})\\ &= (n-q)^{-1}f(D)^{\mathrm{T}}\left\{A^{-1}-A^{-1}H\left(H^{\mathrm{T}}A^{-1}H\right)^{-1}H^{\mathrm{T}}A^{-1}\right\}f(D).\end{aligned}$$

This is the first part of the emulator as discussed in *DiscGPBasedEmulator*. The emulator is formally completed by a second part comprising the posterior distribution of δ, which has density given by

$$\pi^*_\delta(\delta) \propto \pi_\delta(\delta) \times |\widehat{\Sigma}|^{-(n-q)/2}|A|^{-r/2}|H^{\mathrm{T}}A^{-1}H|^{-r/2}.$$

In order to compute the emulator predictions and other tasks, three approaches can be considered.

1. Exact computations require a sample from the posterior distribution of δ. This can be obtained by MCMC; a suitable reference can be found below.
2. A common approximation is simply to fix δ at a single value estimated from the posterior distribution. The usual choice is the posterior mode, which can be found as the value of δ for which $\pi^*_\delta(\delta)$ is maximised. See the page on alternative estimators of correlation hyperparameters (*AltEstimateDelta*).
3. An intermediate approach first approximates the posterior distribution by a multivariate lognormal distribution and then uses a sample from this distribution, as described in the procedure page *ProcApproxDeltaPosterior*.

Each of these approaches results in a set of values (or just a single value in the case of the second approach) of δ, which allow the emulator predictions and other required inferences to be computed.

Although it represents an approximation that ignores the uncertainty in δ, approach 2 has been widely used. It has often been suggested that, although uncertainty in these correlation hyperparameters can be substantial, taking proper account of that uncertainty through approach 1 does not lead to appreciable differences in the resulting emulator. On the other hand, although this may be true if a good single estimate for δ is used, this is not necessarily easy to find, and the posterior mode may sometimes be a poor choice. Approach 3 has not been used much, but can be recommended when there is concern about using just a single δ estimate. It is simpler than the full MCMC approach 1, but should capture the uncertainty in δ well.

8.36.5 Additional Comments

Several computational issues can arise in implementing this procedure. These are discussed in *DiscBuildCoreGP*.

8.36.6 References

Here are two leading textbooks on MCMC:

- Gilks, W.R., Richardson, S. & Spiegelhalter, D.J. (1996). Markov Chain Monte Carlo in Practice. Chapman & Hall.
- Gamerman, D. and Lopes, H. F. (2006). Markov Chain Monte Carlo: Stochastic Simulation for Bayesian Inference. CRC Press.

Although MCMC for the distribution of δ has been reported in a number of articles, they have not given any details for how to do this, assuming instead that the reader is familiar with MCMC techniques.

Details of the linear mean weak prior case can be found in:

Conti, S. and O'Hagan, A. (2009). Bayesian emulation of complex multi-output and dynamic computer models. *Journal of Statistical Planning and Inference.* doi: 10.1016/j.jspi.2009.08.006

The multi-output emulator with the linear mean form is a special case of the outer product emulator. The following reference gives formulae which exploit separable structures in both the mean and covariance functions to achieve computational efficiency that allows very large (output dimension) simulators to be emulated.

J.C. Rougier (2008), Efficient Emulators for Multivariate Deterministic Functions, *Journal of Computational and Graphical Statistics*, 17(4), 827-843. doi:10.1198/106186008X384032.

8.37 Procedure: Build Gaussian process emulator with derivative information

8.37.1 Description and Background

The preparation for building a *Gaussian process* (GP) *emulator* with derivative information involves defining the prior mean and covariance functions, identifying prior distributions for *hyperparameters*, creating a *design* for the *training sample*, then running the *adjoint*, or *simulator* and a method to obtain derivatives, at the input configurations specified in the design. All of this is described in the variant thread on emulators with derivative information (*ThreadVariantWithDerivatives*). The procedure here is for taking those various ingredients and creating the GP emulator.

8.37.2 Additional notation for this page

Including derivative information requires further notation than is specified in the Toolkit notation page (*MetaNotation*). We declare this notation here but it is only applicable to the derivatives thread variant pages.

- The tilde symbol () placed over a letter denotes derivative information and function output combined.
- We introduce an extra argument to denote a derivative. We define $\tilde{f}(x,d)$ to be the derivative of $f(x)$ with respect to input d and so $d \in \{0,1,\ldots,p\}$. When $d=0$ we have $\tilde{f}(x,0)=f(x)$. For simplicity, when $d=0$ we adopt the shorter notation so we use $f(x)$ rather than $\tilde{f}(x,0)$.
- An input is denoted by a superscript on x, while a subscript on x refers to the point in the input space. For example, $x_i^{(k)}$ refers to input k at point i.

8.37.3 Inputs

- GP prior mean function $m(\cdot)$, differentiable and depending on hyperparameters β
- GP prior correlation function $c(\cdot,\cdot)$, twice differentiable and depending on hyperparameters δ
- Prior distribution $\pi(\cdot,\cdot,\cdot)$ for β, σ^2 and δ where σ^2 is the process variance hyperparameter
- We require a design. In the core thread *ThreadCoreGP* the design, D, is an ordered set of points $D = \{x_1, x_2, \ldots, x_n\}$, where each x is a location in the input space. Here, we need a design which in addition to specifying the location of the inputs, also determines at which points we require function output and at which points we require first derivatives. We arrange this information in the design $\tilde{D} = \{(x_k, d_k)\}$, where $k = \{1,\ldots,\tilde{n}\}$ and $d_k \in \{0,1,\ldots,p\}$. We have x_k which refers to the location in the design and d_k determines whether at point x_k we require function output or a first derivative w.r.t one of the inputs. Each x_k is not distinct as we may have a derivative and the function output at point x_k or we may require a derivative w.r.t several inputs at point x_k.
- Output vector is $\tilde{f}(\tilde{D})$ of length $\tilde{n}$.

8.37.4 Outputs

A GP-based emulator in one of the forms discussed in *DiscGPBasedEmulator*.

In the case of general prior mean and correlation functions and general prior distribution:

- A GP posterior conditional distribution with mean function $m^*(\cdot)$ and covariance function $v^*(\cdot,\cdot)$ conditional on $\theta = \{\beta, \sigma^2, \delta\}$. If we want to emulate the derivatives rather than the function output see ProcBuildGPEmulateDerivs.
- A posterior representation for θ

In the case of linear mean function, general correlation function, weak prior information on β, σ^2 and general prior distribution for δ:

- A *t process* posterior conditional distribution with mean function $m^*(\cdot)$, covariance function $v^*(\cdot,\cdot)$ and degrees of freedom b^* conditional on δ
- A posterior representation for δ

As explained in *DiscGPBasedEmulator*, the “posterior representation” for the hyperparameters is formally the posterior distribution for those hyperparameters, but for computational purposes this distribution is represented by a sample of hyperparameter values. In either case, the outputs define the emulator and allow all necessary computations for tasks such as prediction of the simulator output, *uncertainty analysis* or *sensitivity analysis*.

8.37.5 Procedure

General case

We define the following arrays (following the conventions set out in *MetaNotation* where possible).

$\tilde{e} = \tilde{f}(\tilde{D}) - \tilde{m}(\tilde{D})$, an $\tilde{n} \times 1$ vector, where $\tilde{m}(x, 0) = m(x)$, and $\tilde{m}(x, d) = \frac{\partial}{\partial x^{(d)}} m(x)$ if $d \neq 0$.

$\tilde{A} = \tilde{c}(\tilde{D}, \tilde{D})$, an $\tilde{n} \times \tilde{n}$ matrix, where the exact form of $\tilde{c}(\cdot,\cdot)$ depends on where derivatives are included. The general expression for this is: $\tilde{c}(.,.) = \mathrm{Corr}\{\tilde{f}(x_i, d_i), \tilde{f}(x_j, d_j)\}$ and we can break it down into three cases:

- Case 1 is for when $d_i = d_j = 0$ and as such represents the covariance between 2 points. This is the same as in *ThreadCoreGP* and is given by:

$$\mathrm{Corr}\{\tilde{f}(x_i, 0), \tilde{f}(x_j, 0)\} = c(x_i, x_j).$$

- Case 2 is for when $d_i \neq 0$ and $d_j = 0$ and as such represents the covariance between a derivative and a point. This is obtained by differentiating $c(\cdot,\cdot)$ w.r.t input d_i:

$$\mathrm{Corr}\{\tilde{f}(x_i, d_i), \tilde{f}(x_j, 0)\} = \frac{\partial c(x_i, x_j)}{\partial x_i^{(d_i)}}, \text{for } d_i \neq 0.$$

- Case 3 is for when $d_i \neq 0$ and $d_j \neq 0$ and as such represents the covariance between two derivatives. This is obtained by differentiating $c(\cdot,\cdot)$ twice: once w.r.t input d_i and once w.r.t input d_j:

$$\mathrm{Corr}\{\tilde{f}(x_i, d_j), \tilde{f}(x_j, d_j)\} = \frac{\partial^2 c(x_i, x_j)}{\partial x_i^{(d_i)} \partial x_j^{(d_j)}}, \text{for } d_i, d_j \neq 0.$$

  - Case 3a. If $d_i, d_j \neq 0$ and $d_i = d_j$ we have a special version of Case 3 which gives:

$$\mathrm{Corr}\{\tilde{f}(x_i, d_i), \tilde{f}(x_j, d_i)\} = \frac{\partial^2 c(x_i, x_j)}{\partial x_i^{(d_i)}, x_j^{(d_i)}}, \text{for } d_i \neq 0.$$

$\tilde{t}(x) = \tilde{c}\{\tilde{D}, (x, 0)\}$, an $\tilde{n} \times 1$ vector function of x. We have $d = 0$ as here we want to emulate function output. To emulate derivatives, $d \neq 0$ and this is covered in the generic thread on emulating derivatives (*ThreadGenericEmulateDerivatives*).

Then, conditional on θ and the training sample, the output vector $\tilde{f}(x,0) = f(x)$ is a multivariate GP with posterior mean function

$$m^*(x) = m(x) + \tilde{t}(x)^{\mathrm{T}} \tilde{A}^{-1} \tilde{e}$$

and posterior covariance function

$$v^*(x_i,x_j) = \sigma^2 \{ c(x_i,x_j) - \tilde{t}(x_i)^{\mathrm{T}} \tilde{A}^{-1} \tilde{t}(x_j) \}.$$

This is the first part of the emulator as discussed in *DiscGPBasedEmulator*. The emulator is completed by a second part formally comprising the posterior distribution of θ, which has density given by

$$\pi^*(\beta,\sigma^2,\delta) \propto \pi(\beta,\sigma^2,\delta) \times (\sigma^2)^{-\tilde{n}/2} |\tilde{A}|^{-1/2} \times \exp\{-\tilde{e}^{\mathrm{T}} \tilde{A}^{-1} \tilde{e}/(2\sigma^2)\}.$$

For the output vector $\tilde{f}(x,d)$ with $d \neq 0$ see the procedure page on building an emulator of derivatives (*ProcBuildEmulateDerivsGP*).

Linear mean and weak prior case

Suppose now that the mean function has the linear form $m(x) = h(x)^{\mathrm{T}}\beta$, where $h(\cdot)$ is a vector of q known *basis functions* of the inputs and β is a $q \times 1$ column vector of hyperparameters. When $d \neq 0$ we therefore have $\tilde{m}(x,d) = \tilde{h}(x,d)^{\mathrm{T}}\beta = \frac{\partial}{\partial x^{(d)}} h(x)^{\mathrm{T}}\beta$. Suppose also that the prior distribution has the form $\pi(\beta,\sigma^2,\delta) \propto \sigma^{-2}\pi_\delta(\delta)$, i.e. that we have weak prior information on β and σ^2 and an arbitrary prior distribution $\pi_\delta(\cdot)$ for δ.

Define $\tilde{A}$ and $\tilde{t}(x)$ as in the previous case. In addition, define the $\tilde{n} \times q$ matrix

$$\tilde{H} = [\tilde{h}(x_1,d_1), \ldots, \tilde{h}(x_{\tilde{n}},d_{\tilde{n}})]^{\mathrm{T}},$$

the vector

$$\widehat{\beta} = \left(\tilde{H}^{\mathrm{T}} \tilde{A}^{-1} \tilde{H} \right)^{-1} \tilde{H}^{\mathrm{T}} \tilde{A}^{-1} \tilde{f}(\tilde{D})$$

and the scalar

$$\widehat{\sigma}^2 = (\tilde{n}-q-2)^{-1} \tilde{f}(\tilde{D})^{\mathrm{T}} \left\{ \tilde{A}^{-1} - \tilde{A}^{-1}\tilde{H} \left(\tilde{H}^{\mathrm{T}} \tilde{A}^{-1} \tilde{H} \right)^{-1} \tilde{H}^{\mathrm{T}} \tilde{A}^{-1} \right\} \tilde{f}(\tilde{D}).$$

Then, conditional on δ and the training sample, the output vector $\tilde{f}(x,0) = f(x)$ is a t process with $b^* = \tilde{n} - q$ degrees of freedom, posterior mean function

$$m^*(x) = h(x)^{\mathrm{T}}\widehat{\beta} + \tilde{t}(x)^{\mathrm{T}} \tilde{A}^{-1} (\tilde{f}(\tilde{D}) - \tilde{H}\widehat{\beta})$$

and posterior covariance function

$$v^*(x_i,x_j) = \widehat{\sigma}^2 \{ c(x_i,x_j) - \tilde{t}(x_i)^{\mathrm{T}} \tilde{A}^{-1} \tilde{t}(x_j) + \left(h(x_i)^{\mathrm{T}} - \tilde{t}(x_i)^{\mathrm{T}} \tilde{A}^{-1} \tilde{H} \right) \left(\tilde{H}^{\mathrm{T}} \tilde{A}^{-1} \tilde{H} \right)^{-1} \left(h(x_j)^{\mathrm{T}} - \tilde{t}(x_j)^{\mathrm{T}} \tilde{A}^{-1} \tilde{H} \right)^{\mathrm{T}} \}.$$

This is the first part of the emulator as discussed in *DiscGPBasedEmulator*. The emulator is formally completed by a second part comprising the posterior distribution of δ, which has density given by

$$\pi^*_\delta(\delta) \propto \pi_\delta(\delta) \times (\widehat{\sigma}^2)^{-(\tilde{n}-q)/2} |\tilde{A}|^{-1/2} |\tilde{H}^{\mathrm{T}} \tilde{A}^{-1} \tilde{H}|^{-1/2}.$$

In order to derive the sample representation of this posterior distribution for the second part of the emulator, three approaches can be considered.

1. Exact computations require a sample from the posterior distribution of δ. This can be obtained by MCMC; a suitable reference can be found below.

2. A common approximation is simply to fix δ at a single value estimated from the posterior distribution. The usual choice is the posterior mode, which can be found as the value of δ for which $\pi^*(\delta)$ is maximised. See the alternatives page on estimators of correlation hyperparameters (*AltEstimateDelta*).
3. An intermediate approach first approximates the posterior distribution by a multivariate lognormal distribution and then uses a sample from this distribution, as described in the procedure page *ProcApproxDeltaPosterior*.

Each of these approaches results in a set of values (or just a single value in the case of the second approach) of δ, which allow the emulator predictions and other required inferences to be computed.

Although it represents an approximation that ignores the uncertainty in δ, approach 2 has been widely used. It has often been suggested that, although uncertainty in these correlation hyperparameters can be substantial, taking proper account of that uncertainty through approach 1 does not lead to appreciable differences in the resulting emulator. On the other hand, although this may be true if a good single estimate for δ is used, this is not necessarily easy to find, and the posterior mode may sometimes be a poor choice. Approach 3 has not been used much, but can be recommended when there is concern about using just a single δ estimate. It is simpler than the full MCMC approach 1, but should capture the uncertainty in δ well.

8.37.6 References

Morris, M. D., Mitchell, T. J. and Ylvisaker, D. (1993). Bayesian design and analysis of computer experiments: Use of derivatives in surface prediction. *Technometrics*, 35, 243-255.

8.38 Procedure: Data Pre-Processing and Standardisation

In this page we describe the process of pre-processing data, which might often be undertaken prior to for example *screening* or more general *emulation*. This can take several forms. A very common pre-processing step is centring, which produces data with zero mean. If the range of variation is known a priori a simple linear transformation to the range [0,1] is often used. It might also be useful to standardise (sometimes called normalise) data to produce zero mean and unit variance. For multivariate data it can be useful to whiten (or sphere) the data to have zero mean and identity covariance, which for one variable is the same as standardisation. The linear transformation and normalisation processes are not equivalent since the latter is a probabilistic transformation using the first two moments of the observed data.

8.38.1 Centring

It is often useful to remove the mean from a data set. In general the mean, $\mathrm{E}[x]$, will not be known and thus must be estimated and the centered data is given by: $x' = x - \mathrm{E}[x]$. Centring will often be used if a zero mean Gaussian process is being used to build the emulator, although in general it would be better to include an explicit mean function in the emulator.

8.38.2 Linear transformations

To linearly transform the data region $x \in [c, d]$ to another domain $x' \in [a, b]$:

$x' = \frac{x-c}{d-c}(b-a) + a$

In experimental design the convention is for $[a, b] = [0, 1]$.

8.38.3 Standardising

If the domain of the design region is not known, samples from the design space can be used to rescale the data to have 0 mean, unit variance by using the process of standardisation. If on the other hand the design domain is known we can employ a linear rescaling.

The process involves estimating the mean $\mu = \mathrm{E}[x]$ and standard deviation of the data σ and applying the transformation $x' = \frac{x-\mu}{\sigma}$. It is possible to standardise each input/output separately which rescales the data, but does not render the outputs uncorrelated. This might be useful in situations where correlations or covariances are difficult to estimate, or where these relationships want to be preserved, so that individual inputs can still be distinguished.

8.38.4 Sphering/Whitening

For multivariate inputs and outputs it is possible to whiten the data, that is convert the data to zero mean, identity variance. This process is a linear transformation of the data and is described in more detail, including a discussion of how to treat a mean function in the procedure page *ProcOutputsPrincipalComponents*, and would typically be applied to outputs rather than inputs.

The data sphering process involves estimating the mean $\mathrm{E}[x]$ and variance matrix of the data $\mathrm{Var}[x]$, computing the eigen decomposition $P\Delta P^T$ of $\mathrm{Var}[x]$ and applying the transformation $x' = P\Delta^{-1/2}P^T(x - \mathrm{E}[x])$. Alternative approaches are possible and are discussed in *ProcOutputsPrincipalComponents*.

8.39 Procedure: Iterate the single step emulator using an exact simulation approach

8.39.1 Description and Background

This page is concerned with task of *emulating* a *dynamic simulator*, as set out in the variant thread on dynamic emulation (*ThreadVariantDynamic*).

We have an emulator for the *single step function* $w_t = f(w_{t-1}, a_t, \phi)$, and wish to predict the full time series $w_1, \ldots, w_T$ for a specified initial *state variable* w_0, time series of *forcing variables* $a_1, \ldots, a_T$ and simulator parameters ϕ. It is not possible to derive analytically a distribution for $w_1, \ldots, w_T$ if $f(\cdot)$ is modelled as a *Gaussian Process*, so here we use simulation to sample from the distribution of $w_1, \ldots, w_T$. We simulate a large number of series $w_1^{(i)}, \ldots, w_T^{(i)}$ for $i = 1, \ldots, R$, and then use the simulated series for making predictions and reporting uncertainty.

8.39.2 Inputs

- An emulator for the single step function $w_t = f(w_{t-1}, a_t, \phi)$, formulated as a GP or *t-process* conditional on hyperparameters, *training inputs* $D = \{x_1, \ldots, x_n\}$ and training outputs $f(D)$.
- A set $\{\theta^{(1)}, \ldots, \theta^{(s)}\}$ of emulator hyperparameter values.
- The initial value of the state variable w_0.
- The values of the forcing variables $a_1, \ldots, a_T$.
- The values of the simulator parameters ϕ.
- Number of realisations R required.

For notational convenience, we suppose that $R \leq s$. For discussion of the choice of R, including the case $R > s$, see the discussion page on Monte Carlo estimation (*DiscMonteCarlo*).

8.39.3 Outputs

- A set of simulated state variable time series $w_1^{(i)},\dots,w_T^{(i)}$ for $i=1,\dots,R$

8.39.4 Procedure

Note: methods for simulating outputs from Gaussian process emulators are described in the procedure page *ProcOutputSample*.

A single time series $w_1^{(i)},\dots,w_T^{(i)}$ can be generated as follows.

1. Using the emulator with hyperparameters $\theta^{(i)}$, sample from the distribution of $f(w_0,a_1,\phi)$ to obtain $w_1^{(i)}$. Then iterate the following steps 2-4 for $t=1,\dots,T-1$.
2. Construct a new training dataset of inputs $(D,D^{(i,t)})$, where $D^{(i,t)} = \{(w_0,a_1,\phi),(w_1^{(i)},,a_2,\phi),\dots,(w_{t-1}^{(i)},a_t,\phi)\}$ and outputs $\{f(D),w_1^{(i)},\dots,w_t^{(i)}\}$. To clarify, training inputs and outputs are paired as follows:

$$\text{Training inputs:}\left(\begin{array}{c} x_1\\ \vdots\\ x_n\\ (w_0,a_1,\phi)\\ (w_1^{(i)},a_2,\phi)\\ \vdots\\ (w_{t-1}^{(i)},a_t,\phi)\end{array}\right)\qquad \text{Training outputs:}\left(\begin{array}{c} f(x_1)\\ \vdots\\ f(x_n)\\ w_1^{(i)}\\ w_2^{(i)}\\ \vdots\\ w_t^{(i)}\end{array}\right)$$

3. Re-build the single step emulator given the new training data defined in step 2. It may be necessary to thin the new training data first before building the emulator. The set of inputs $(D,D^{(i,t)})$ may contain points close together, which can make inversion of $A=c\{(D,D^{(i,t)}),(D,D^{(i,t)})\}$ difficult. See discussion in Additional Comments.
4. Sample from the distribution of $f(w_t^{(i)},a_{t+1},\phi)$ to obtain $w_{t+1}^{(i)}$

The whole process is repeated to obtain R simulated time series $w_1^{(i)},\dots,w_T^{(i)}$ for $i=1,\dots R$. The sample $w_1^{(i)},\dots,w_T^{(i)}$ for $i=1,\dots R$ is a sample from the joint distribution of $w_1,\dots,w_T$ given the emulator training data and $w_0,a_1,\dots,a_T,\phi$.

8.39.5 Additional Comments

As commented in step 3, computational difficulties can arise if the training set of inputs $(D,D^{(i,t)})$ contains inputs that are too close together. This is likely to occur, as $(w_{t+1}^{(i)},a_{t+2})$ is likely to be close to $(w_t^{(i)},a_{t+1})$. This is problem is not unique to the use of dynamic emulators, and is discussed in the page on computational issues in building a GP emulator (*DiscBuildCoreGP*). A strategy that has been used with some success for this procedure is to consider the emulator variance of $f(w_t^{(i)},a_{t+1},\phi)$ given training inputs $(D,D^{(i,t)})$ and outputs $\{f(D),w_1^{(i)},\dots,w_t^{(i)}\}$, and only add the new training input $(w_t^{(i)},a_{t+1},\phi)$ and associated output to the training data at iteration $t+1$ if the variance of $f(w_t^{(i)},a_{t+1},\phi)$ is sufficiently large. If the variance is very small, so that the emulator already 'knows' the value of $f(w_t^{(i)},a_{t+1},\phi)$, then adding this point to the training data will little effect on the distribution of $f(\cdot)$.

8.40 Procedure: Exchange Algorithm

8.40.1 Description and Background

The exchange algorithm has been used and modified by several authors to construct a D-optimal and other types of design. Fedorov (1972), Mitchell and Miller (1970), Wynn (1970) , Mitchell (1974) and Atkinson and Donev (1989) study versions of this algorithm. The main idea of all versions is start with an initial feasible design and then greedily modify the design by exchange until a satisfactory design is obtained. The following steps are the general steps for an exchange algorithm.

8.40.2 Inputs

1. A large candidate set, such as a full factorial design (scaled integer grid) or a spacefilling design.
2. A initial design chosen, at random, or preferably, a space filling design.
3. An objective function M, which is based on an optimal design criterion.

8.40.3 Outputs

1. The optimal or near optimal design $D = \{x_1, x_2, \cdots, x_n\}$.
2. The value of the target function at the obtained design

8.40.4 Procedure

1. Start with the initial n point design.
2. Compute the target function M.
3. General step. Find a set of points of size k from the current design and replace with another set of k points chosen from the candidate set. If the value of M, after this exchange, is improved, keep the new design and update M. The set of points removed from the current design can be put back in the candidate set.
4. Repeat till the stopping rule applies. It may be that after many attempts at a good exchange there is no improvement or only a small improvement in the design. As for global optimisation algorithms it is usual to plot the value of the objective function against the number of improving exchanges, or simply the number of exchanges.

8.40.5 Additional Comments, References, and Links

1. Note that the algorithm will need an updating rule for the objective function which is preferable fast to compute. There is in principle no restriction on the objective function.
2. There are are many variations. One version is to first add k optimally and sequentially to obtain a design with $n + k$ points and then remove k optimally and sequentially. Such operation is called an *excursion*. The choice of the test points outside the current design at step 3 (above) may be computational slow in which case a fast strategy is to choose them at random. The same idea can be applied when we use an excursion method, at least for the forward part of the excursion.
3. The method can be used in Bayesian or classical optimal design.

A.C. Atkinson and A.N. Donev. The construction of exact D-optimal designs with application to blocking response surface designs. *Biometrika*, 76, 515-526, 1989.

H.P. Wynn. The sequential generation of D-optimal experimental designs, *Ann. Math. Stat.*, 41, 1644-1655, 1970.

N. K. Nguyen and A.J. Miller, A review of exchange algorithms for constructing descrete D-optimal designs, *J. of Computational Statistics & Data Analysis* 14, 489-498, 1992.

T.J. Mitchell,An algorithm for the construction of D-optimal designs, *Technometrics*, 20, 203-210, 1974.

T.J. Mitchell, and A. J. Miller, Use of 'design repair' to construct designs for special linear models, *Math. Div. Ann. Progr. Rept.*,130-131, 1970.

V. Fedorov. *Theory of Optimal Experiments*. Academic Press, New York ,1972.

8.41 Procedure: Explore the full simulator design region to identify a suitable single step function design region

8.41.1 Description and Background

This page is concerned with task of *emulating* a *dynamic simulator*, as set out in the variant thread on dynamic emulation (*ThreadVariantDynamic*).

We have an emulator for the *single step function* $w_t = f(w_{t-1}, a_t, \phi)$ given an initial assessment of the input region of interest $\mathcal{X}_{single}$, and some training data. However, the 'correct' region $\mathcal{X}_{single}$ depends on the input region of interest $\mathcal{X}_{full}$ for the full simulator, and the single step function $f(\cdot)$. Here, we use simulation to make an improved assessment of $\mathcal{X}_{single}$ given $\mathcal{X}_{full}$ and an emulator for $f(.)$.

8.41.2 Inputs

- An emulator for the single step function $w_t = f(w_{t-1}, a_t, \phi)$, formulated as a *GP* or *t-process* conditional on hyperparameters, *training inputs* D and training outputs $f(D)$.
- A set $\{\theta^{(1)}, \ldots, \theta^{(s)}\}$ of emulator hyperparameter values.
- The input region of interest $\mathcal{X}_{full}$ for the full simulator.

8.41.3 Outputs

- A set of N simulated joint time series $\{(w_0^{(i)}, a_1^{(i)}), \ldots, (w_{T-1}^{(i)}, a_T^{(i)})\}$ for $i = 1, \ldots, N$.

8.41.4 Procedure

1. Choose a set of design points $\{x_1^{full}, \ldots, x_N^{full}\}$ from $\mathcal{X}_{full}$, with $x_i^{full} = (w_0^{(i)}, a_1^{(i)}, \ldots, a_T^{(i)}, \phi^{(i)})$. Both N and the design points can be chosen following the principles in the alternatives page on training sample design (*AltCoreDesign*), but see additional comments at the end. Then for $i = 1, \ldots, N$:
2. For x_i^{full} and $\theta^{(i)}$, generate one random time series $w_1^{(i)}, \ldots, w_T^{(i)}$ using the simulation method given in the procedure page *ProcExactIterateSingleStepEmulator* (use $R = 1$ within this procedure). Note that we have assumed $N \leq s$ here. If it is not possible to increase s and we need $N > s$, then we suggest cycling round the set $\{\theta^{(1)}, \ldots, \theta^{(s)}\}$ for each iteration i.
3. Organise the forcing inputs from x_i^{full} and simulated state variables into a joint time series $(w_{t-1}^{(i)}, a_t^{(i)})$ for $t = 1, \ldots, T$.

8.41.5 Additional Comments

Since generating a single time series $w_1^{(i)}, \dots, w_T^{(i)}$ should be a relatively quick procedure, we can use a larger value of N than might normally be considered in *AltCoreDesign*. The main aim here is to establish the boundaries of $\mathcal{X}_{single}$, and so we should check that any such assessment is stable for increasing values of N.

8.42 Procedure: Fourier Expansion

8.42.1 Description and Background

This procedure is used to find the eigenvalues and the eigenfunctions of the covariance kernel when the analytical solution is not available. The orthonormal basis $\{\theta_i\}$ is used to solve the problem. This procedure uses Fourier basis functions as the set of orthonormal basis functions.

In general, the eigenfunction $\phi_i(t)$ is written as

$$\phi_i(t) = \sum_{k=1}^{M} d_{ik}\theta_k(t) = \theta(t)^T D_i = D_i^T \theta(t),$$

where $\{\theta_i(t)\}$ is the set of orthonormal basis functions and $\{d_{ik}\}$ is the set of unknown coefficients for the expansion.

8.42.2 Inputs

1. The covariance function, see *AltCorrelationFunction*, and estimates of its parameters.
2. p, the number of eigenvalues and eigenfunctions required to truncate the Karhunen Loeve Expansion at.
3. A set of M adequate basis functions, where M is chosen to be odd. The basis functions are written as

$$\begin{aligned} \theta_1(t) &= 1, \\ \theta_2(t) &= \cos(2\pi t), \\ \theta_3(t) &= \sin(2\pi t), \\ \theta_{2i}(t) &= \cos(2\pi i t), \\ \theta_{2i+1}(t) &= \sin(2\pi i t), i = 1, 2, \dots \frac{M-1}{2}. \end{aligned}$$

8.42.3 Output

1. The set of eigenvalues $\{\lambda_i\}, i = 1 \cdots p$.
2. The matrix of unknown coefficients, D.
3. An approximated covariance kernel; $R(s,t) = \sum_{i=1}^{p} \lambda_i \phi_i(s)\phi_i(t) = \phi(s)^T \Lambda \phi(t) = \theta(s)^T D^T \Lambda D \theta(t)$.

8.42.4 Procedure

1. Replace $\phi_i(s)$ in the Fredholm equation $\int_0^1 R(s,t)\phi_i(s)ds = \lambda_i \phi_i(t)$ with $D_i^T \theta(t)$, then we have $D_i^T \int_0^1 R(s,t)\theta(s)ds = D_i^T \lambda_i \theta(t)$.
2. Multiply both sides of $D_i^T \int_0^1 R(s,t)\theta(s)ds = D_i^T \lambda_i \theta(t)$ by $\theta(t)$.

3. Integrate both sides of $D_i^T \int_0^1 R(s,t)\theta(s)\theta(t)^T ds = D_i^T \lambda_i \theta(t)\theta(t)^T$ with respect to t.
4. Define $A = \int_0^1 \int_0^1 R(s,t)\theta(s)\theta(t)dsdt, \quad B = \int_0^1 \theta(t)\theta^T(t)dt$ where A is a symmetric positive definite matrix and B is a diagonal positive matrix.
5. Matrix implemenation. Write the integration in (4) as $D_{p\times M}A_{M\times M} = \Lambda_{p\times p}DB_{M\times M}$ which is equivalent to $AD^T = BD^T\Lambda$.
6. Express B as $B^{\frac{1}{2}}B^{\frac{1}{2}}$.
7. Express the form in (5) as $AD^T = B^{\frac{1}{2}}B^{\frac{1}{2}}D^T\Lambda$.
8. Multilpy both sides of (7) by $B^{-\frac{1}{2}}$, so that $B^{-\frac{1}{2}}AB^{-\frac{1}{2}}B^{\frac{1}{2}}D^T = B^{-\frac{1}{2}}D^T\Lambda$.
9. Assume $E = B^{-\frac{1}{2}}D^T$ then $B^{-\frac{1}{2}}AB^{-\frac{1}{2}}E = E\Lambda$.
10. Solve the eigen-problem of $B^{-\frac{1}{2}}AB^{-\frac{1}{2}}E = E\Lambda$.
11. Compute D using $D = E^TB^{-\frac{1}{2}}$.

8.42.5 Additional Comments, References, and Links

For the spatial case of dimension d, the procedure is repeated d times, and the number of eigenvalues, similarly the number of eigenfunctions, is equal to p^d.

8.43 Procedure: Haar wavelet expansion

This procedure is used to find the eigenvalues and the eigenfunctions of the covariance kernel when the analytical solution is not available. The othonoromal basis function $\{\psi_i\}$ is used to solve the problem. This procedure uses Haar wavelet basis functions as the set of orthonormal basis functions.

The eigenfunction $\phi_i(t)$ is expressed as a linear combination of Haar orthonormal basis functions

$$\phi_i(t) = \sum_{k=1}^{M} d_{ik}\psi_k(t) = \theta(t)^T D_i = D_i^T\psi(t).$$

The Haar wavelet, the simplest wavelet basis function, is defined as

$$\psi(x) = \left\{ \begin{array}{cc} 1 & 0 < x < \frac{1}{2} \\ -1 & \frac{1}{2} \le x < 1 \\ 0 & \text{otherwise} \end{array} \right. .$$

8.43.1 Inputs

1. The covariance function , see *AltCorrelationFunction*, and the estimates of its parameters.
2. p the number of eigenvalues and eigenfunctions required to truncate the Karhunen Loeve Expansion at.
3. $M = 2^n$ orthogonal basis functions on $[0,1]$ constructed in the following way

$$\begin{aligned} \psi_1 &= 1 \\ \psi_i &= \psi_{j,k}(x) \\ i &= 2^j + k + 1 \\ j &= 0, 1, \cdots, n-1 \\ k &= 0, 1, \cdots, 2^j - 1 \end{aligned}$$

where

$$\psi(x) = \begin{cases} 1 & k2^{-j} < x < 2^{-j-1} + k2^{-j} \\ -1 & 2^{-j-1} + k2^{-j} \leq x < 2^{-j} + k2^{-j} \\ 0 & \text{otherwise} \end{cases}$$

8.43.2 Output

1. The eigenvalues λ_i , $i = 1 \cdots p$.
2. The matrix of unknown coefficients D.
3. An approximated covariance function; $R(s,t) = \Psi(s)^T D^T \Lambda D \Psi(t)$.

8.43.3 Procedure

1. Write the eigenfunction as $\phi_i(t) = \sum_{k=1}^{M} d_{ik}\psi_k(t) = \Psi^T(t)D_i$ so that, $R(s,t) = \sum_{m=1}^{M}\sum_{n=1}^{M} a_{mn}\psi_m(s)\psi_n(t) = \Psi(s)^T A\Psi(t)$.
2. Choose M time points such that $t_j = \frac{2i-1}{2M}, 1 \leq i \leq M$.
3. Compute the covariance function C for those M points.
4. Apply the 2D wavelet transform (discrete wavelet transform) on C to obtain the matrix A.
5. Substitute $R(s,t) = \Psi(s)^T A\Psi(t)$ in $\lambda_i\phi_i(t) = \int_0^1 R(s,t)\phi_i(s)$, then we have $\lambda_i\Psi^T(t)D_i = \Psi^T(t)AHD_i$.
6. Define H as a diagonal matrix with diagonal elements $h_{11} = 1, h_{ii} = 2^{-j}$ $i = 2^j + k + 1, \quad j = 0, 1, \cdots, n-1$ and $k = 0, 1, \cdots, 2^j - 1$.
7. Define the whole problem as $\Lambda_{p\times p} D_{p\times M}\Psi(t)_{M\times 1} = D_{p\times M}H_{M\times M}A_{M\times M}\Psi(t)_{M\times 1}$.
8. From (7), we have $\Lambda D = DHA$.
9. Multiply both sides of (8) by $H^{\frac{1}{2}}$
10. Express the eigen-problem as $\Lambda DH^{\frac{1}{2}} = DH^{\frac{1}{2}}H^{\frac{1}{2}}AH^{\frac{1}{2}}$ or $\Lambda\hat{D} = \hat{D}.\hat{A}$ where $\hat{D} = DH^{\frac{1}{2}}$ and $\hat{A} = H^{\frac{1}{2}}AH^{\frac{1}{2}}$.
11. Solve the eigen-problem in (10), then $\Phi(t) = D\Psi(t) = \hat{D}H^{-\frac{1}{2}}\Psi(t)$.

8.43.4 Additional Comments, References, and Links

The application of the Haar wavelet procedure shows a better approximation and faster implementation than the Fourier procedure given in *ProcFourierExpansionForKL*. For one dimension implementation shows that $2^5 = 32$ provides very accurate and quite fast approximations.

8.44 Procedure: Generate a Halton design

8.44.1 Description and Background

A Halton design is one of a number of non-random space-filling designs suitable for defining a set of points in the *simulator* input space for creating a *training sample*.

The n point Halton design in p dimensions is generated by a generator set $g = (g_1, \ldots, g_p)$ of prime numbers. See the "Additional Comments" below for discussion of the choice of generators.

8.44.2 Inputs

- Number of dimensions p
- Number of points desired n
- Set of prime generators $g_1, \dots, g_p$

8.44.3 Outputs

- Halton design $D = \{x_1, x_2, \dots, x_n\}$

8.44.4 Procedure

1. For each $i = 1, 2, \dots, p$ and $j = 1, 2, \dots, n$, let a_{ij} be the representation in prime base g_i of the number j. Let n_{ij} be the number of digits used for a_{ij} (i.e. n_{ij} is the logarithm to base g_i of j, rounded up to the nearest integer), and let R_{ij} be the result of reversing the digits of a_{ij} and evaluating this as a number in base g_i.
2. For each $i = 1, 2, \dots, p$ and $j = 1, 2, \dots, n$, let $x_{ij} = R_{ij}/g_i^{n_{ij}}$.
3. For $j = 1, 2, \dots, n$, the j-th design point is $x_j = (x_{1j}, x_{2j}, \dots, x_{pj})$.

For example, if $j = 10$ and $g_i = 2$, then $a_{ij} = 1010$, from which we have $n_{ij} = 4$. Then R_{ij} is the binary number 0101, i.e 5, so that $x_{ij} = 5/2^4 = 0.3125$.

8.44.5 Additional Comments

A potential problem with Halton designs is the difficulty in finding suitable generators. One suggestion is to let g_i be the i-th prime, but this may not work well when p is large.

8.44.6 References

The following is a link to the repository for Matlab code for the Halton sequence in up to 11 dimensions: CPHalton-Sequence.m (*disclaimer*).

8.45 Procedure: Generate a Latin hypercube

8.45.1 Description and Background

A Latin hypercube (LHC) is a random set of points in $[0, 1]^p$ constructed so that for $i = 1, 2, \dots, p$ the i-th coordinates of the points are spread evenly along [0,1].

8.45.2 Inputs

- Number of dimensions p
- Number of points desired n

8.45.3 Outputs

- LHC $D = \{x_1, x_2, \dots, x_n\}$

8.45.4 Procedure

1. For each $i = 1, 2, \dots, p$, independently generate n random numbers $u_{i1}, u_{i2}, \dots, u_{in}$ in [0,1] and a random permutation $b_{i1}, b_{i2}, \dots, b_{in}$ of the integers $0, 1, \dots, n-1$.
2. For each $i = 1, 2, \dots, p$ and $j = 1, 2, \dots, n$ let $x_{ij} = (b_{ij} + u_{ij})/n$.
3. For $j = 1, 2, \dots, n$, the j-th LHC point is $x_j = (x_{1j}, x_{2j}, \dots, x_{pj})$.

8.45.5 Additional Comments

The construction of a LHC implies that the projection of the set of points into the i-th dimension results in n points that are evenly spread over that dimension. If we project into two or more of the p dimensions, a LHC may also appear to be well spread (space-filling) in that projection, but this is not guaranteed.

8.46 Procedure: Generate a lattice design

8.46.1 Description and Background

A lattice design is one of a number of non-random space-filling designs suitable for defining a set of points in the *simulator* input space for creating a *training sample*.

The n point lattice in p dimensions is generated by a positive integer generator set $g = (g_1, \dots, g_p)$. See the "Additional Comments" below for discussion of the choice of generators.

8.46.2 Inputs

- Number of dimensions p
- Number of runs desired n
- Set of positive integer generators $g_1, \dots, g_p$

8.46.3 Outputs

- Lattice design $D = \{x_1, x_2, \dots, x_n\}$

8.46.4 Procedure

For $j = 0, \dots, n-1$, generate lattice points as

$$x_{j+1} = \left(\frac{j}{n} g_1 \mathrm{mod} 1, \dots, \frac{j}{n} g_d \mathrm{mod} 1 \right).$$

Note that the operator "mod 1" here has the effect of returning the fractional part of each number. For instance, if $j = 7, n = 50$ and $g_1 = 13$, then

$$\frac{j}{n}g_1 = 1.82$$

and so

$$\frac{j}{n}g_1 \mathrm{mod} 1 = 0.82.$$

8.46.5 Additional Comments

A potential problem with lattice designs is the difficulty in finding suitable generators for a lattice. A condition for generators is that $g_1, \ldots, g_d$ and n should form a set of relatively prime numbers. However, this seems to be a necessary but not sufficient condition to obtain a lattice that fills the design space well.

8.46.6 References

Bates, R.A., Riccomagno, E., Schwabe, R., Wynn, H. (1998). The use of lattices in the design of high-dimensional experiments. IMS Lecture Notes, 34, 26-35.

Sloan, I.H., Joe, S. (1994). Lattice methods for multiple integration. Clarendon Press, Oxford.

Matlab code for generating lattice designs is available from Ron Bates (*disclaimer*).

8.47 Procedure: Sampling the posterior distribution of the correlation lengths

8.47.1 Description and Background

This procedure shows how to draw samples from the posterior distribution of the correlation lengths $\pi^*_\delta(\delta)$ and how to use the drawn samples to make predictions using the emulator. This procedure complements the procedure for building a Gaussian process emulator for the core problem (*ProcBuildCoreGP*), and represents the formal way of accounting for the uncertainty about the true value of the correlation lengths δ.

8.47.2 Inputs

- An emulator as defined in *ProcBuildCoreGP*, using a linear mean and a weak prior.

8.47.3 Outputs

- A set of s samples for the correlation lengths δ, denoted as $\tilde{\delta}$.
- A posterior mean $\tilde{m}^*(\cdot)$ and covariance function $\tilde{u}^*(\cdot, \cdot)$, conditioned on the drawn samples $\tilde{\delta}$.

8.47.4 Procedure

A method for drawing samples from the posterior distribution of δ is via the Metropolis-Hastings algorithm. Setting up this algorithm requires an initial estimate of the correlation lengths, and a proposal distribution. An initial estimate can be the value that maximises the posterior distribution, as this is defined in the discussion page for finding the posterior mode of correlation lengths (*DiscPostModeDelta*). We call this estimate $\hat{\delta}$.

According to reference [1], a proposal distribution can be

$$\delta^{(i)} \sim \mathcal{N}(\delta^{(i-1)}, V)$$

where $\delta^{(i-1)}$ is the sample drawn at the $(i-1)$-th step of the algorithm, and

$$V = -\frac{2.4}{\sqrt{p}}\left(\frac{\partial^2 \pi^*_\delta(\hat{\delta})}{\partial \hat{\delta}}\right)^{-1}$$

The Hessian matrix $\frac{\partial^2 \pi^*_\delta(\hat{\delta})}{\partial \hat{\delta}}$ is the same as $\frac{\partial^2 g(\tau)}{\partial \tau_l \partial \tau_k}$, which was defined in *DiscPostModeDelta*, after substituting the derivatives $\partial A/\partial \tau$ with the derivatives $\partial A/\partial \delta$. The latter are given by

$$\left(\frac{\partial A}{\partial \delta_k}\right)_{i,j} = A(i,j)\left[\frac{2(x_{k,i}-x_{k,j})^2}{\delta_k^3}\right]$$
$$\left(\frac{\partial^2 A}{\partial^2 \delta_k}\right)_{i,j} = A(i,j)\frac{(x_{k,i}-x_{k,j})^2}{\delta_k^4}\left[\frac{4(x_{k,i}-x_{k,j})^2}{\delta_k^2}-6\right]$$

and finally

$$\left(\frac{\partial^2 A}{\partial \delta_l \partial \delta_k}\right)_{i,j} = A(i,j)\left[\frac{2(x_{l,i}-x_{l,j})^2}{\delta_l^3}\right]\left[\frac{2(x_{k,i}-x_{k,j})^2}{\delta_k^3}\right]$$

Having defined the initial estimate of δ and the proposal distribution $\mathcal{N}(\delta^{(i-1)}, V)$ we can set up the following Metropolis Hastings algorithm

1. Set $\delta^{(1)}$ equal to $\hat{\delta}$
2. Add to $\delta^{(i)}$ a normal variate drawn from $\mathcal{N}(0, V)$ and call the result δ'
3. Calculate the ratio $\alpha = \frac{\pi^*_\delta(\delta')}{\pi^*_\delta(\delta^{(i)})}$
4. Draw w from a uniform distribution in [0,1]
5. if $w < \alpha$ set $\delta^{(i+1)}$ equal to δ', else set it equal to $\delta^{(i)}$
6. Repeat steps 2-5 s times

Finally, if we define as $m^{*(i)}(x)$ and $u^{*(i)}(x,x')$ the posterior mean and variance of the emulator using sample $\delta^{(i)}$, a total estimate of these two quantities, taking into account all the s samples of δ drawn, is given by

$$\tilde{m}^*(x) = \frac{1}{s}\sum_{i=1}^{s} m^{*(i)}(x)$$

and

$$\tilde{u}^*(x,x') = \frac{1}{s}\sum_{i=1}^{s} u^{*(i)}(x,x') + \frac{1}{s}\sum_{i=1}^{s}\left[m^{*(i)}(x)-\tilde{m}^*(x)\right]\left[m^{*(i)}(x')-\tilde{m}^*(x')\right]$$

The procedure for predicting the simulator outputs using more than one hyperparameter sets is described in greater detail in page (*ProcPredictGP*).

8.47.5 References

1. Gilks, W.R., Richardson, S. & Spiegelhalter, D.J. (1996). Markov Chain Monte Carlo in Practice. Chapman & Hall.

8.48 Procedure: Morris screening method

The Morris method, also know as the Elementary Effect (EE) method, utilises a discrete approximation of the average value of the Jacobian (matrix of the partial derivatives of the *simulator* output with respect to each of the simulator inputs) of the simulator over the input space. The motivation for the method was *screening* for *deterministic* computer models with moderate to large numbers of inputs. The method relies on a one-factor-at-a-time (OAT) experimental *design* where the effects of a single factor on the output is assessed sequentially. The individual randomised experimental designs are known as trajectories. The method's main advantage is the lack of assumptions on the inputs and functional dependency of the output to inputs such as monotonicity or linearity.

8.48.1 Algorithm

The algorithm involves generating R trajectory designs, as described below. Each trajectory design is used to compute the expected value of the elementary effects in the simulator function locally. By averaging these a global approximation is obtained. The steps are:

1. Rescale all input variables to operate on the same scale using either standardisation or linear scaling as shown in the procedure page for data standardisation (*ProcDataPreProcessing*). Otherwise different values of the step size (see below) will be needed for each input variable.
2. Each point in the trajectory differs from the previous point in only one input variable by a fixed step size, Δ. For k variables, each trajectory has $k+1$ points, changing each variable exactly once. The start point for the trajectory is random, although this can be modified to improve the algorithm. See the discussion below.
3. Compute the elementary effect for each input variable $1,\ldots,k$: $EE_i(x) = \frac{f(x+\Delta e_i)-f(x)}{\Delta}$. e_i is the unit vector in the direction of the i^{th} axis for $i = 1,\ldots,k$. Each elementary effect is computed with observations at the pair of points $x, x+\Delta e_i$ that differ in the i^{th} input variable by the fixed step size Δ.
4. Compute the moments of the elementary effects distribution for each input variable:

$$\mu_i = \sum_{r=1}^{R} \frac{EE_i(x_r)}{R},$$

$$\mu_i^* = \sum_{r=1}^{R} \left|\frac{EE_i(x_r)}{R}\right|,$$

$$\sigma_i = \sqrt{\sum_{r=1}^{R} \frac{(EE_i(x_r)-\mu_i)^2}{R}}.$$

The sample moment μ_i is an average effect measure, and a high value suggests a dominant contribution of the i^{th} input factor in positive or negative response values (i.e. typically linear, or at least monotonic). The sample moment μ_i^* is a total effect measure; a high value indicates large influence of the corresponding input factor. μ_i may prove misleading due to cancellation effects (that is on average over the input space the output goes up in response to the input as much as it comes down), thus to capture main effects μ_i^* should be used. Non-linear and interaction effects are estimated with σ_i. The total number of model runs needed in the Morris's method is $(k+1)R$.

An effects plot is constructed by plotting μ_i or μ_i^* against σ_i. This plot is a visual tool to detecting and ranking effects.

An example on synthetic data demonstrating the Morris method is provided at the example page *ExamScreeningMorris*.

8.48.2 Setting the parameters of the Morris method

There is interest in undertaking input screening with as few simulator runs as possible, but as the number of input factors k is fixed, the size of the experiment required is controlled by the number of trajectory designs R. Usually small values of R are used; for instance, in Morris (1991) the values $R = 3$ and $R = 4$ were used in the examples. A value of R between 10 and 50 is mentioned in the more recent literature (see References). A larger value of R may improve the quality of the global estimates at the price of extra runs. For a reasonably high dimensional input space, with more than say 10 inputs, it would seem unwise to select R less than 10, since coverage of the space, and thus global effects estimates require something close to space filling. It is likely for large k the number of trajectory designs will need some dependency on R.

The step size Δ is selected in such a way that all the simulator runs lie in the input space and the elementary effects are computed with reasonable precision. The usual choice of Δ in the literature is determined by the input space considered for experimentation, which is a k dimensional grid constructed with p uniformly spaced values for each input. The number p is recommended to be even and Δ to be an integer multiple of $1/(p-1)$. Morris (1991) suggests a value of $\Delta = p/2(p-1)$ that ensures good coverage of the input space with few trajectories. One value for Δ is generally used for all the inputs, but the method can be generalised to instead use different values of Δ and p for every input.

8.48.3 Extending the Morris method

In Morris's original proposal, the starting points of the trajectory designs were taken at random from the input space grid. Campolongo (2007) proposed generating a large number of trajectories, selecting the subset that maximise the distance between trajectories in order to cover the design space. Another option is to use a *Latin Hypercube design* or a *Sobol* sequence to select the starting points of the trajectories.

A potential drawback of OAT runs in the Morris's method is that design points fall on top of each other when projected into lower dimensions. This disadvantage becomes more apparent when the design runs are to be used in further modelling after discarding unimportant factors. An alternative is to construct a randomly rotated simplex at every point from which elementary effects are computed (Pujol, 2009). The computation of distribution moments μ_i, μ_i^*, σ_i and further analysis is similar as the Morris's method, with the advantage that projections of the resulting design do not fall on top of existing points, and all observations can be reused in a later stage. A potential disadvantage of this approach is the loss of efficiency in the computation of elementary effects.

Lastly, it is possible to modify the standard Morris algorithm to minimize the number of simulator runs required by employing a sequential version of the algorithm. Details can be found in Boukouvalas et al (2010).

8.48.4 References

Morris, M. D. (1991, May). Factorial sampling plans for preliminary computational experiments. *Technometrics*, 33 (2), 161–174.

Boukouvalas, A., Gosling, J.P. and Maruri-Aguilar, H., An efficient screening method for computer experiments. NCRG Technical Report, Aston University (2010)

Saltelli, A., Chan, K. and Scott, E. M. (eds.) (2000). Sensitivity Analysis. Wiley.

Francesca Campolongo, Jessica Cariboni, and Andrea Saltelli. An effective screening design for sensitivity analysis of large models. *Environ. Model. Softw.*, 22(10):1509–18, 2007.

Francesca Campolongo, Jessica Cariboni, Andrea Saltelli, and W. Schoutens. Enhancing the Morris Method. In *Sensitivity Analysis of Model Output*, pages 369–79, 2004.

Gilles Pujol. Simplex-based screening designs for estimating metamodels. *Reliability Engineering & System Safety*, 94:1156–60, 2009.

8.49 Procedure: Generate an optimised Latin hypercube design

8.49.1 Description and Background

A Latin hypercube (LHC) is a random set of points in $[0, 1]^p$ constructed so that for $i = 1, 2, \dots, p$ the i-th coordinates of the points are spread evenly along [0,1]. However, a single random LHC will rarely have good enough space-filling properties in the whole of $[0, 1]^p$ or satisfy other desirable criteria. Therefore, it is usual to generate many LHCs and then select the one having the best value of a suitable criterion.

We present here the procedure for a general optimality criterion and also include details of the most popular criterion, known as maximin. A thorough consideration of optimality criteria may be found in the discussion page on technical issues in training sample design for the core problem (*DiscCoreDesign*).

8.49.2 Inputs

- Number of dimensions p
- Number of points desired n
- Number of LHCs to be generated N
- Optimality criterion $C(D)$

8.49.3 Outputs

- Optimised LHC design $D = \{x_1, x_2, \dots, x_n\}$

8.49.4 Procedure

Procedure for a general criterion

1. For $k = 1, 2, \dots, N$ independently, generate a random LHC D_k using the procedure page on generating a Latin hypercube (*ProcLHC*) and evaluate the criterion $C_k = C(D_k)$.
2. Let $K = \arg\max\{D_k\}$ (i.e. K is the number of the LHC with the highest criterion value).
3. Set $D = D_K$.

Note that the second step here assumes that high values of the criterion are desirable. If low values are desirable, then change argmax to argmin.

Maximin criterion

A commonly used criterion is

$$C(D) = \min_{j \neq j'} |x_j - x_{j'}|,$$

where $|x_j - x_{j'}|$ denotes a measure of distance between the two points x_j and $x_{j'}$ in the design. The distance measure is usually taken to be squared Euclidean distance: that is, if $u = (u_1, u_2, \ldots, u_p)$ then we define

$$|u| = u^T u = \sum_{i=1}^{p} u_i^2.$$

High values of this criterion are desirable.

8.49.5 Additional Comments

Note that the resulting design will not truly be optimal with respect to the chosen criterion because only a finite number of LHCs will be generated.

8.50 Procedure: Generate random outputs from an emulator at specified inputs

8.50.1 Description and Background

This procedure describes how to randomly sample output values from an *emulator*. This, of course, requires a fully probabilistic emulator such as the *Gaussian process* emulator described in the core thread *ThreadCoreGP*, rather than a *Bayes linear* emulator. You should refer to *ThreadCoreGP*, the procedure page on building a GP emulator (*ProcBuildCoreGP*), the variant thread on analysing a simulator with multiple outputs using GP methods (*ThreadVariantMultipleOutputs*) and the procedure page on building multivariate GP emulators (*ProcBuildMultiOutputGP*) for definitions of the various functions used in this procedure.

8.50.2 Inputs

- An emulator
- Emulator hyperparameters θ
- A set of points $D = (x'_1, \ldots, x'_{n'})$ in the input space at which randomly sampled outputs are required.

8.50.3 Outputs

- A set of jointly sampled values from the distribution of $f(D') = \{f(x'_1), \ldots, f(x'_{n'})\}$.

8.50.4 Procedure

Scalar output general case

1. Define

$$m^*(D') = \{m^*(x'_1), \ldots, m^*(x'_{n'})\}^{\mathrm{T}}.$$

2. Define $v^*(D', D')$, the $n' \times n'$ matrix with i, j element given by $v^*(x'_i, x'_j)$.
3. Generate the random vector $\{f(x'_1), \ldots, f(x'_{n'})\}^{\mathrm{T}}$ from $N_{n'}\{m^*(D'), v^*(D', D')\}$, the multivariate normal distribution with mean vector $m^*(D')$ and variance matrix $v^*(D', D')$.

Multivariate output general case

We have a simulator $f(.)$ that produces a vector $f(x)$ of r outputs for a single input value x. To clarify notation, we generate the random vector $F=\{f(x'_1),\ldots,f(x'_{n'})\}^{\rm T}$, arranged as follows:

$$F=\left(\begin{array}{c}\text{(output 1 at input } x'_1\text{)}\\ \vdots\\ \text{(output 1 at input } x'_{n'}\text{)}\\ \vdots\\ \text{(output r at input } x'_1\text{)}\\ \vdots\\ \text{(output r at input } x'_{n'}\text{)}\end{array}\right).$$

1. Define

   $$m^*(D')=\{m^*(x'_1)^{\rm T},\ldots,m^*(x'_{n'})^{\rm T}\}^{\rm T}.$$

   In the multivariate case this is an $n'\times r$ matrix with each of the r columns representing one of the simulator outputs. We arrange this into an $n'r\times 1$ column vector by stacking the r columns of the $n'\times r$ matrix into a single column vector (the vec operation):

   $$\mu^*(D')=\mathrm{vec}\{m^*(D')\}.$$

2. Define V, the $(n'r)\times(n'r)$ matrix with i,j element defined as the posterior covariance between elements i and j of $F=\{f(x'_1),\ldots,f(x'_{n'})\}^{\rm T}$.
3. Generate the random vector F from $N_{n'r}\{\mu^*(D'),V\}$, the multivariate normal distribution with mean vector $\mu^*(D')$ and variance matrix :math:‘ V‘.

Multivariate output general case with separable covariance function

Suppose we have a *separable* covariance function of the form $\Sigma c(\cdot,\cdot)$, where Σ is the output space covariance matrix, and $c(\cdot,\cdot)$ is the input space covariance function. Following the notation in *ProcBuildMultiOutputGP*, we write the posterior covariance function as

$$\mathrm{Cov}[f(x),f(x')|f(D)]=\Sigma c^*(x,x')=\Sigma\{c(x,x')-c(x)^{\rm T}A^{-1}c(x)\},$$

with $A=c(D,D)$. The posterior variance matrix :math:‘ V‘ of $\{f(x'_1),\ldots,f(x'_{n'})\}^{\rm T}$ can then be written as

$V=\Sigma\otimes c^*(D',D')$, where $\otimes$ is the kronecker product. We can now generate $\{f(x'_1),\ldots,f(x'_{n'})\}^{\rm T}$ using the matrix normal distribution:

1. Define U_Σ to be the lower triangular square root of Σ
2. Define U_c to be the lower triangular square root of $c^*(D',D')$
3. Generate $Z_{n',r}$: an $n'\times r$ matrix of independent standard normal random variables
4. The random draw from the distribution of $f(D')$ is given by $F=U_cZ_{n',r}U_\sigma$

F is an $n'\times r$ matrix, arranged as follows:

$$F=\left(\begin{array}{ccc}\text{(output 1 at input } x'_1\text{)} & \cdots & \text{(output r at input } x'_1\text{)}\\ \vdots & & \vdots\\ \text{(output 1 at input } x'_{n'}\text{)} & \cdots & \text{(output r at input } x'_{n'}\text{)}\end{array}\right).$$

Scalar output linear mean and weak prior case

1. Define

$$m^*(D') = \{m^*(x'_1), \ldots, m^*(x'_{n'})\}^{\mathrm{T}}.$$

2. Define $v^*(D', D')$, the $n' \times n'$ matrix with i, j element given by $v^*(x'_i, x'_j)$.
3. Generate the random vector $\{f(x'_1), \ldots, f(x'_{n'})\}^{\mathrm{T}}$ from a multivariate student t distribution with mean vector $m^*(D')$, variance matrix $V^*(D', D')$, and $n - q$ degrees of freedom.

   As an alternative to sampling from the multivariate student t distribution, you can first first sample a random value of σ^2 and then sample from a multivariate normal instead:

   a. Sample τ^2 from the $\Gamma\{(n-q)/2, (n-q-2)\hat{\sigma}^2/2\}$ distribution. Note the parameterisation of the gamma distribution here: if $W \sim Gamma(a, b)$ then the density function is $p(w) = \frac{b^a}{\Gamma(a)} w^{a-1} \exp(-bw)$.

   b. Set $\sigma^2 = 1/\tau^2$ and replace $\hat{\sigma}^2$ by $1/\tau^2$ in the formula for $v^*(x'_i, x'_j)$.

   c. Compute $v^*(D', D')$, the $n' \times n'$ matrix with i, j element given by $v^*(x'_i, x'_j)$.

   d. Generate the random vector $\{f(x'_1), \ldots, f(x'_{n'})\}^{\mathrm{T}}$ from $N_{n'}\{m^*(D'), v^*(D', D')\}$, the multivariate normal distribution with mean vector $m^*(D')$ and variance matrix $v^*(D', D')$.

Multivariate output linear mean and weak prior case

We have a simulator $f(\cdot)$ that produces a vector $f(x)$ of r outputs for a single input value x. To clarify notation, we generate the random vector $F = \{f(x'_1), \ldots, f(x'_{n'})\}^{\mathrm{T}}$, arranged as follows:

$$F = \begin{pmatrix} \text{(output 1 at input } x'_1) \\ \vdots \\ \text{(output 1 at input } x'_{n'}) \\ \vdots \text{output r at input } x'_1) \\ \vdots \\ \text{(output r at input } x'_{n'}) \end{pmatrix}.$$

1. Define

$$m^*(D') = \{m^*(x'_1)^{\mathrm{T}}, \ldots, m^*(x'_{n'})^{\mathrm{T}}\}^{\mathrm{T}}.$$

   In the multivariate case this is an $n' \times r$ matrix with each of the r columns representing one of the simulator outputs. We arrange this into an $n'r \times 1$ column vector by performing the Vec operation:

$$\mu^*(D') = \mathrm{vec}\{m^*(D')\}.$$

2. Define V, the $(n'r) \times (n'r)$ matrix with i, j element defined as the posterior covariance between elements i and j of $F = \{f(x'_1), \ldots, f(x'_{n'})\}^{\mathrm{T}}$.
3. Generate the random vector F from a multivariate student t distribution with mean vector $\mu^*(D')$, variance matrix V, and $n - q$ degrees of freedom.

Multivariate output linear mean, weak prior, and separable covariance function case

Suppose we have a *separable* covariance function of the form $\Sigma c(\cdot,\cdot)$, where Σ is the output space covariance matrix, and $c(\cdot,\cdot)$ is the input space covariance function. Following the notation in *ProcBuildMultiOutputGP*, we write the posterior covariance function as

$$\textrm{Cov}[f(x),f(x')|f(D)] = \widehat{\Sigma}c^*(x,x') = \widehat{\Sigma}\left\{c(x,x') - c(x)^{\rm T} A^{-1} c(x') + R(x)\left(H^{\rm T} A^{-1} H\right)^{-1} R(x')^{\rm T}\right\},$$

with $A=c(D,D)$ and $R(x)=h(x)^{\rm T}-c(x)^{\rm T}A^{-1}H$. The posterior variance matrix V of $\{f(x'_1),\ldots,f(x'_{n'})\}^{\rm T}$ can then be written as

$V=\widehat{\Sigma}\otimes c^*(D',D')$, where $\otimes$ is the kronecker product. We can now generate $\{f(x'_1),\ldots,f(x'_{n'})\}^{\rm T}$ using the matrix student t distribution:

1. Define U_Σ to be the lower triangular square root of $\widehat{\Sigma}$
2. Define U_c to be the lower triangular square root of $c^*(D',D')$
3. Generate a $n'\times r$ matrix $T_{n',r}$ having a multivariate t distribution with uncorrelated elements, in the following three sub-steps.

   a. Generate $Z_{n',r}$: an $n'\times r$ matrix of independent standard normal random variables.

   b. Generate $W\sim Gamma\{(n-q)/2, 0.5\}$.

   c. Set $T_{n',r}=\frac{1}{\sqrt{W/(n-q)}}Z_{n',r}$.

   Note the parameterisation of the gamma distribution here: if $W\sim Gamma(a,b)$ then the density function is $p(w)=\frac{b^a}{\Gamma(a)}w^{a-1}\exp(-bw)$.

4) The random draw from the distribution of $f(D')$ is given by $F=U_cT_{n',r}U_\sigma$

   F is an $n'\times r$ matrix, arranged as follows:

$$F=\begin{pmatrix} (\text{output 1 at input } x'_1) & \cdots & (\text{output r at input } x'_1) \\ \vdots & & \vdots \\ (\text{output 1 at input } x'_{n'}) & \cdots & (\text{output r at input } x'_{n'}) \end{pmatrix}.$$

8.51 Procedure: Transformed outputs

8.51.1 Description and Background

It is sometimes appropriate to build an *emulator* of some *transformation* of the *simulator* output of interest, rather than the output itself. See the discussion page on the Gaussian assumption (*DiscGaussianAssumption*) for the background to using output transformations.

The emulator allows inference statements about the transformed output, and for instance can be used to conduct *uncertainty analysis* or *sensitivity analysis* of the transformed output. However, the interest lies in making such inferences and analyses on the original, untransformed output. The procedure explains how to construct these from a fully *Bayesian* emulator of the transformed output.

In the case of a *Bayes linear* emulator, entirely different methods are needed to make inferences about the original output.

8.51.2 Inputs

The input is a fully Bayesian emulator for the transformed simulator output.

We will use the following notation. For any given input configuration x, let the original output be $f(x)$, and let the transformed output be $t(x) = g\{f(x)\}$, so that g denotes the transformation. The emulator therefore provides a probability distribution for $t(x)$ at any or all values of x. We suppose that the transformation is one-to-one and that the inverse transformation is g^{-1}, i.e. $f(x) = g^{-1}\{t(x)\}$.

8.51.3 Outputs

Outputs are any desired inferences about properties of the original output. For instance, if we let the inputs be random, denoting them now by X, then uncertainty analysis of $f(X)$ might include as one specific inference the expectation (with respect to the *code uncertainty*) of the uncertainty mean $M = \mathrm{E}[f(X)]$. In this case the property is M and the inference is the mean (interpreted as an estimate of M).

8.51.4 Procedure

The simplest procedure is to use simulation. The method of *simulation based inference* for emulators requires only a little modification. The method involves drawing random realisations from the emulator distribution, and then computing the property in question for each such realisation. The set of property values so derived is a sample from the (code uncertainty) distribution of that parameter. From this sample we compute the necessary inferences.

When we have a transformed output, we add one more step. We draw random realisations from the emulator for $t(x)$, but we now apply the inverse transformation g^{-1} to every point on the realisation before computing the property value. This ensures that the parameter values are now a sample from the distribution of that property as defined for the original output.

8.51.5 Additional Comments

There are specific cases where we can do better than this. For some transformations we can derive the distribution of $f(x)$ for any given x analytically, at least conditionally on *hyperparameters*. In some cases, we may even be able to derive uncertainty analysis or sensitivity analysis. This is an area for ongoing research.

8.52 Procedure: Principal components and related transformations of simulator outputs

8.52.1 Description and Background

One of the methods for *emulating* several outputs from a *simulator* considered in the alternatives page on approaches to emulating multiple outputs (*AltMultipleOutputsApproach*) is to transform the outputs into a set of 'latent outputs' that can be considered uncorrelated, and then to emulate the latent outputs separately. We consider here various linear transformations for this purpose, although probably the most useful is the principal components transformation.

There are two steps in the procedure.

1. Obtain an $r \times r$ variance matrix V for the r outputs.
2. Derive an $r \times r$ transformation matrix P and apply the transformation.

We consider each of these steps in outline before describing the procedure itself.

The variance matrix V for the outputs is unknown and must be estimated. The basic source for such estimation is the *training sample*, comprising output vectors from the simulator at n points in the space of possible input configurations.

It is simple to compute the sample variance matrix of these vectors, but sample covariances and correlations measure correlation between the deviations of pairs of outputs from their sample means, whereas for V we require correlation in deviations from the mean *functions* $m_u(\cdot)(u = 1, 2, \ldots, r)$ of the r outputs. The procedure below therefore involves first estimating these mean functions.

For the second step, there are various ways to derive a linear transformation for a row vector of random quantities w with variance matrix V so that the elements of the transformed row vector $w^\star = wP^\mathrm{T}$ are uncorrelated. In general, the covariance matrix of $w^\star$ is $B = PVP^\mathrm{T}$, and we seek a matrix P for which this becomes diagonal. Then the elements of $w^\star$ are uncorrelated with variances equal to the diagonal elements of B.

Probably the most useful for transforming simulator outputs is the principal components transformation. Here we let P be the matrix of eigenvectors of V, whereupon B is the diagonal matrix with the corresponding eigenvalues of V down the diagonal.

Another class of transformations arise by letting $P = S^{-1}$, where S is a square root of V in the sense that $V = SS^\mathrm{T}$. Then $B = I$, the $r \times r$ identity matrix. One square root matrix that is easy to calculate (and to invert to get P) is the Cholesky square root; see also the procedure page on Pivoted Cholesky decomposition (*ProcPivotedCholesky*). However, it is easy to see that if S is a square root of V then so is RS for any orthogonal matrix R.

8.52.2 Inputs

- The $(r \times 1)$ row vectors $f(x_i)$ (for $i = 1, 2, \ldots n$) of outputs from the simulator at n design points $D = \{x_1, x_2, \ldots, x_n\}$.

8.52.3 Outputs

- Transformation matrix P.
- Transformed 'latent' output vectors $f^\star(x_i)$ (for $i = 1, 2, \ldots n$) at the n design points.

8.52.4 Procedure

Derive V

1. Choose a set of *basis functions* in the form of a $q \times 1$ vector function $h(\cdot)$. The same set of basis functions should be used for all the outputs, so if a certain basis function is thought *a priori* to be appropriate to model the expected behaviour of any one of the outputs (as a function of the inputs) then this should be included in $h(\cdot)$.
2. For each $u = 1, 2, \ldots, r$, construct the $n \times 1$ vector F_u of values of output u. Thus, F_u is the u-th column of the matrix $f(D)$. (See the toolkit notation page (*MetaNotation*) for the use of D as a function argument, as in $f(D)$.)
3. For each u, fit a conventional linear regression model to vector F_u, with $h(D)$ as the usual $n \times q$ X-matrix. Let the $q \times 1$ vector $\hat{\beta}_u$ be the fitted (i.e. estimated) regression coefficients.
4. For each u, form the $n \times 1$ vector of residuals $E_u = F_u - h(D)^\mathrm{T}\hat{\beta}_u$ and form these columns into the $n \times r$ residual matrix E.
5. Set $V = n^{-1}E^\mathrm{T}E$.

The fitting of linear regression models is available in all the major statistical software packages and in many programming languages. An alternative to fitting the conventional linear regression model would be to build an emulator for each output using the core thread that treats the core problem using Gaussian methods (*ThreadCoreGP*) and to define $\hat{\beta}$ as in the procedure page for building a Gaussian process emulator (*ProcBuildCoreGP*); however, although this might in principle be better it is unlikely in practice to make an appreciable difference or to be worth the extra effort.

Derive transformation

Having obtained V, the transformation matrix P and the diagonal variance matrix B can be obtained using standard software to apply the relevant decomposition method. For instance, software for eigenvalue decomposition is available in numerous computing packages (e.g. Matlab), the Cholesky decomposition is almost as widely available and the pivoted Cholesky decomposition is detailed in *ProcPivotedCholesky*.

The latent outputs are then characterised by the (row) vectors $f^\star(x_i) = f(x_i)P^{\mathrm{T}}$, for $i = 1, 2, \ldots, n$. Equivalently, $f^\star(D) = f(D)P^{\mathrm{T}}$.

8.52.5 Additional Comments

Transformations are a very general idea, and there are related uses in the toolkit. The procedure page on transforming outputs (*ProcOutputTransformation*) concerns transforming individual outputs (generally in a nonlinear way), motivated principally by making the assumption of a Gaussian process more appropriate.

Principal components and related transformations, particularly the pivoted Cholesky decomposition, are used in *validation*; see the procedure page on validating a Gaussian process emulator (*ProcValidateCoreGP*). The variance matrix V is in that case obtained directly from the emulator as the predictive variance matrix of the validation sample.

If the outputs are strongly correlated, then it might be possible to reduce the number of outputs, for example by retaining only the first $r^\star < r$ eigenvectors of the P matrix with the largest eigenvalues, although in this case it is necessary to include an additional nugget effect as discussed in the alternatives page on emulator prior correlation function (*AltCorrelationFunction*).

8.53 Procedure: Pivoted Cholesky decomposition

8.53.1 Description and Background

The Cholesky decomposition is named after Andre-Louis Cholesky, who found that a symmetric positive-definite matrix can be decomposed into a lower triangular matrix and the transpose of the lower triangular matrix. Formally, the Cholesky method decomposes a symmetric positive definite matrix A uniquely into a product of a lower triangular matrix L and its transpose, i.e. $A = LL^T$, or equivalently $A = R^T R$ where R is a upper triangular matrix.

The Pivoted Cholesky decomposition, or the Cholesky decomposition with complete pivoting, of a matrix A returns a permutation matrix P and the unique upper triangular matrix R such that $P^T AP = R^T R$. The permutation matrix is an orthogonal matrix, so the matrix A can be rewritten as $A = (RP^T)^T RP^T$

An arbitrary permutation of rows and columns of matrix A can be decomposed by the Cholesky algorithm, $P^T AP = R^T R$, where P is a permutation matrix and R is an upper triangular matrix. The permutation matrix is an orthogonal matrix, so the matrix A can be rewritten as $A = PR^T RP^T$, so that $C = PR^T$ is the pivoted Cholesky decomposition of A.

Pivoted Cholesky algorithm: This algorithm computes the Pivoted Cholesky decomposition $P^T AP = R^T R$ of a symmetric positive semidefinite matrix $A \in \Re^{n \times n}$. The nonzero elements of the permutation matrix P are given by $P(piv(k), k) = 1, k = 1, \ldots, n$

More details about the numerical analysis of the pivoted Cholesky decomposition can be found in Higham (2002).

8.53.2 Inputs

- Symmetric positive-definite matrix A

8.53.3 Outputs

- Permutation matrix P
- Unique upper triangular matrix R

8.53.4 Procedure

```
**Initialise**
$R = \text{zeros}(\text{size}(A))$
$piv(k) = k, \forall k \in [1, n]$
**Repeat** for $k = 1 : n$
    {Finding the pivot}
      $B = A(k : n, k : n)$
      $l = \{i : A(i, i) == \max diag\,(B)\}$
    {Swap rows and columns}
      $A(:, k) <=> A(:, l)$
      $R(:, k) <=> R(:, l)$
      $A(k, :) <=> A(l, :)$
      $piv(k) <=> piv(l)$
    {Cholesky decomposition}
      $R(k, k) = \sqrt{A(k, k)}$
      $R(k, k + 1 : n) = R(k, k)^{-1} A(k, k + 1 : n)$
    {Updating $A$}
      $A(k + 1 : n, k + 1 : n) = A(k + 1 : n, k + 1 : n) - R(k, k + 1 : n)^T R(k, k + 1 : n)$
**End repeat**
```

8.53.5 Additional Comments

If A is a variance matrix, the pivoting order given by the permutation matrix P has the following interpretation: the first pivot is the index of the element with the largest variance, the second pivot is the index of the element with the largest variance conditioned on the first element, the third pivot is the index of the element with the largest variance conditioned on the first two elements element, and so on.

8.53.6 References

- Higham, N. J. Accuracy and Stability of Numerical Algorithms. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2002. ISBN 0-89871-521-0. Second Edition.

8.54 Procedure: Predict simulator outputs using a GP emulator

8.54.1 Description and Background

A *Gaussian process* (GP) *emulator* is a statistical representation of knowledge about the outputs of a *simulator* based on the Gaussian process as a probability distribution for an unknown function. The unknown function in this case is the simulator, viewed as a function that takes inputs and produces one or more outputs. One use for the emulator is to predict what the simulator would produce as output when run at one or several different points in the input space. This procedure describes how to derive such predictions in the case of a GP emulator built with the procedure for a single output (*ProcBuildCoreGP*) or the procedure for multiple outputs (*ProcBuildMultiOutputGP*) . The multiple output case will be emphasised only where this differs from the single output case.

8.54.2 Inputs

- An emulator
- A single point x' or a set of points $x'_1, x'_2, \ldots, x'_{n'}$ at which predictions are required for the simulator output(s)

8.54.3 Outputs

- Predictions in the form of statistical quantities, such as the expected value of the output(s), the variance (matrix) of the outputs, the probability that an output exceeds some threshold, or a sample of values from the predictive distribution of the output(s)

8.54.4 Procedure

The emulator, as developed for instance in *ProcBuildCoreGP* or *ProcBuildMultiOutputGP*, has two parts. The first is a distribution, either a GP or its cousin the *t-process*, for the simulator output function conditional on some hyperparameters. The second is generally a collection of sets of values of the hyperparameters being conditioned on, but may be just a single set of values. We let s denote the number of hyperparameter sets provided with the emulator. When we have a single set of hyperparameter values, $s = 1$. See the discussion page on the forms of GP based emulators (*DiscGPBasedEmulator*) for more details.

The procedure for computing predictions generally takes the form of computing the appropriate predictions from the GP or t-process, given that the hyperparameters takes each of the s sets of values in turn, and if $s > 1$ then combining the resulting s predictions. See also the discussion page on Monte Carlo estimation (*DiscMonteCarlo*), where this approach is treated in more detail and in particular there is consideration of how many sets of hyperparameters should be used.

Predictive mean

The conditional mean of the output at x', given the hyperparameters, is obtained by evaluating the mean function of the GP or t-process at that point. When $s = 1$, this is done with the hyperparameters fixed at the single set of values. When $s > 1$, we evaluate the mean function using each of the s hyperparameter sets and the predictive mean is the average of those s conditional means.

If required for a set of points $x'_1, x'_2, \ldots, x'_{n'}$, the predictive mean of the output vector is the vector of predictive means obtained by applying the above procedure to each x'_j separately. In the multi-ouput case things are somewhat more complicated. Each output is itself a $1 \times r$ vector, so the outputs at several input configurations can be treated in two ways, either as as vector of vectors (that is a $n' \times r$ matrix) or more helpfully stacked into a $n'r \times 1$ vector of outputs at each input configuration. This vector can then be treated as described above.

Predictive variance (matrix)

Consider the case where we wish to derive the predictive variance matrix of the output vector at a set of points $x'_1, x'_2, \ldots, x'_{n'}$. If $s = 1$ the predictive variance matrix is just the matrix of conditional variances and covariances from the GP or t process, using the single set of hyperparameter values.

If $s > 1$ the procedure is more complex, and requires several steps as follows.

1. For $i = 1, 2, \ldots, s$ fix the hyperparameters at the i-th set, and given these values of the hyperparameters, compute the vector of conditional means E_i and the matrix V_i of conditional variances and covariances from the GP or t-process.
2. Compute the average values $\bar{E} = s^{-1}\sum_{i=1}^{s} E_i$ and $\bar{V} = s^{-1}\sum_{i=1}^{s} V_i$. (Note that $\bar{E}$ is the predictive mean described above.)
3. Compute the variance matrix of the conditional means, $W = s^{-1}\sum_{i=1}^{s}(E_i - \bar{E})(E_i - \bar{E})^{\mathrm{T}}$.
4. The predictive variance matrix is $\bar{V} + W$.

Prediction at a single point x' is the special case $n' = 1$ of this procedure. In brief, the predictive variance is either just the conditional variance evaluated with the single set of hyperparameter values, or if $s > 1$ the average of the conditional variances plus the variance of the conditional means. To handle the multi-output case the most simple approach is to pursue the vectorisation of the outputs to a $n'r \times 1$ vector. In this case the variance matrix is $n'r \times n'r$ (with a range of possible simplifications if the covariance is assumed to be *separable*. This (potentially very large) variance matrix can be treated identically to the single output case.

Probability of exceeding a threshold

The conditional probability of exceeding a threshold can be computed from the GP or t-process for any given set of hyperparameter values. For a GP, this means computing the probability of exceeding a given value for a normal random variable with given mean and variance. For the t-process it is the probability of exceeding that value for a t random variable with given mean, variance and degrees of freedom. For $s > 1$, the predictive probability is the average of the conditional probabilities.

For multiple outputs this is more complex, since it is possible to ask more complex questions, such as the joint probability of two or more outputs exceeding some threshold. The complexity depends on the assumptions made in constructing the multivariate emulator, and is discussed in the alternatives page on approaches to multiple outputs (*AltMultipleOutputsApproach*). For example if separate independent emulators are used, then the probability of all outputs lying above some threshold will be the product of the individual probabilities of each output being above the threshold. This will not be true if the outputs are correlated and the full multivariate GP or t-process should be used.

Sample of predictions

Suppose we wish to draw a sample of N values from the predictive distribution of the simulator output at the input x', or of the outputs at the points $x'_1, x'_2, \ldots, x'_{n'}$. This means using N sets of hyperparameter values. If $N < s$, then we select a subset of the full set of available hyperparameter sets. (These will usually have been produced by Markov chain Monte Carlo sampling, in which case the subset should be chosen by thinning the sequence of hyperparameter sets, e.g. if $N = s/2$ we could take only even numbered hyperparameter sets.)

If $N > s$ we will need to reuse some hyperparameter sets. Although this is generally undesirable, in the case $s = 1$ it is unavoidable! However, it may be feasible to obtain a larger sample of hyperparameter sets: see *DiscMonteCarlo*.

For each chosen hyperparameter set, we make a *single* draw from the conditional distribution of the output(s) given by the GP or t-process, conditional on that hyperparameter set. Procedures for generating random outputs are described in *ProcOutputSample*.

8.54.5 Additional Comments

It is possible to develop procedures for other kinds of predictions, but not all will be simple. For instance to output a predictive credible interval would be a more complex procedure.

8.55 Procedure: Predicting a function of multiple outputs

8.55.1 Description and Background

When we have a multiple output simulator, we will sometimes be interested in a deterministic function of one or more of the outputs. Examples include:

- A simulator with outputs that represent amounts of rainfall at a number of locations, and we wish to predict the total rainfall over a region, which is the sum of the rainfall outputs at the various locations (or perhaps a weighted sum if the locations represent subregions of different sizes).
- A simulator that outputs the probability of a natural disaster and its consequence in loss of lives, and we are interested in the expected loss of life, which is the product of these two outputs.
- A simulator that outputs atmospheric CO_2 concentration and global temperature, and we are interested in using them to compute the gross primary productivity of an ecosystem.

If we know that we are *only* interested in one particular function of the outputs, then the most efficient emulation method is to build a single output emulator for the output of that function. However, there are situations when it is better to first build a multivariate emulator of the raw outputs of the simulator

- when we are interested in both the raw outputs and one or more functions of the outputs;
- when we are interested in function(s) that depend not just on the raw outputs of the simulator, but also on some other auxiliary variables.

In such situations we build the multivariate emulator by following *ThreadVariantMultipleOutputs*. The multivariate emulator can then be used to predict any function of the outputs, at any set of auxiliary variable values, by following the procedure given here.

We consider a simulator $f(\cdot)$ that has r outputs, and a function of the outputs $g(\cdot)$. The procedure for predicting $g(f(\cdot))$ is based on generating random samples of output values from the emulator, using the procedure *ProcOutputSample*.

8.55.2 Inputs

- A multivariate emulator, which is either a multivariate GP obtained using the procedure *ProcBuildMultiOutputGP*, or a multivariate t-process obtained using the procedure *ProcBuildMultiOutputGPSep*, conditional on hyperparameters.
- s sets of hyperparameter values.
- A single point x' or a set of n' points $x'_1, x'_2, \ldots, x'_{n'}$ at which predictions are required for the function $g(.)$.
- N, the size of the random sample to be generated.

8.55.3 Outputs

- Predictions of $g(\cdot)$ at $x'_1, x'_2, \ldots, x'_{n'}$ in the form of a sample of size N of values from the predictive distribution of $g(\cdot)$.

8.55.4 Procedure

For $j = 1, ..., N$,

1. Pick a set of hyperparameter values at random from the s sets that are available.
2. Generate a $n'r \times 1$ random vector F^j from the emulator, using the procedure set out in the 'Multivariate output general case' section of *ProcOutputSample*.
3. Form the $r \times n'$ matrix M^j such that $\mathrm{vec}[M^{jT}] = F^j$.
4. For $\ell = 1, ..., n'$, let m_ℓ^j be the ℓ and let $G_\ell^j = g(m_\ell)$

The sample is then $\{G^j : j = 1, ..., N\}$, where $G^j = (G_1^j, ..., G_{n'}^j)$.

8.56 Procedure: Predict functions of simulator outputs using multiple independent emulators

8.56.1 Description and Background

Where separate, independent *emulators* have been built for different *simulator* outputs, there is often interest in predicting some function(s) of those outputs. The procedures here are for making such predictions. We assume that, whatever method was used to build each emulator the corresponding toolkit thread also describes how to make predictions for that emulator alone.

The individual emulators may be *Gaussian process* (GP) or *Bayes linear* (BL) emulators, although some of the specific procedures given here will only be applicable to GP emulators.

8.56.2 Inputs

- Emulators for r simulator outputs $f_u(x)$, $u = 1, 2, \ldots, r$
- r' prediction functions $f_w^*(x) = g_w\{f_1(x), \ldots, f_r(x)\}$, $w = 1, 2, \ldots, r'$
- A single point x' or a set of points $x'_1, x'_2, \ldots, x'_{n'}$ at which predictions are required for the prediction function(s)

8.56.3 Outputs

- Predictions in the form of statistical quantities, such as expected values, variances and covariances or a sample of values from the (joint) predictive distribution of the prediction function(s) at the required prediction points

8.56.4 Procedures

The simplest case is when the prediction functions are linear in the outputs. Then

$$f_w^*(x) = a_w + f(x)^T b_w,$$

where a_w is a known constant, $f(x)$ is the vector of r outputs $f_1(x), \ldots, f_r(x)$ and b_w is a known $r \times 1$ vector. It is straightforward to derive means, variances and covariances for the functions $f_w^*(x)$ at the prediction point(s) when the prediction functions are linear. For nonlinear functions, the procedure is to draw a large sample from the predictive distribution and to compute means, variances and covariances from this sample (but note that this is only possible for GP emulators).

Predictive means

Suppose that we have linear prediction functions. Let the $n' \times 1$ predictive mean vector for the u-th emulator at the n' prediction points be m_u. (If we only wish to predict a single point, then this is a scalar.) Let the $n' \times r$ matrix M have columns $m_1, \ldots, m_r$. Then the predictive mean (vector) of $f^*_w(x)$ at the n' prediction points is $a_w 1_{n'} + M\, b_w$, where $1_{n'}$ denotes a vector of n' ones, so that $a_w 1_{n'}$ is a $n' \times 1$ vector with all elements equal to a_w.

Predictive variances and covariances

Suppose that we have linear prediction functions. Let the $n' \times n'$ predictive variance matrix for the u-th emulator at the n' prediction points be V_u. (If we only wish to predict a single point, then this is a scalar.) Let b_{uw} be the u-th element of b_w. Then the variance matrix for $f^*_w(x)$ at the n' prediction points is

$$\sum_{u=1}^{r} b^2_{uw} V_u,$$

and the covariance matrix between $f^*_w(x)$ and $f^*_{w'}(x)$ at those points is

$$\sum_{u=1}^{r} b_{uw} b_{uw'} V_u.$$

Sample of predictions

Suppose we wish to draw a sample of N values from the (joint) predictive distribution of the prediction functions at the input x', or at the points $x'_1, x'_2, \ldots, x'_{n'}$. For GP emulators, such samples can be drawn from the predictive distributions of the individual outputs. Let $f^{(I)}_u(x'_t)$ be the I-th sampled value of $f_u(x)$ at t-th prediction point.

Then the I-th sampled value, $I = 1, 2, \ldots, N$, of $f^*_w(x'_t)$ is $g_w\{f^{(I)}_1(x'_t), \ldots, f^{(I)}_r(x'_t)\}$.

8.57 Procedure: Simulating realisations of an emulator

8.57.1 Description and Background

The key device in *MUCM* is the *emulator*, which is a statistical representation of knowledge about the output(s) of a *simulator*. There are two principal approaches to emulation, the fully *Bayesian* approach and the *Bayes linear* approach. The fully Bayesian approach is often characterised by its representation of the simulator output as a *Gaussian process* (GP), although formally the emulator in this approach is only a GP conditional on various uncertain *hyperparameters*. The emulator proper is the result of averaging over the uncertain values of these hyperparameters. One step in that averaging process may produce a related statistical representation known as a *t-process*.

However, it is not feasible generally to average out all the hyperparameters algebraically, to produce a clean formulation for the emulator proper. Instead, within the MUCM toolkit we formulate the emulator in two parts; see the discussion page on forms of GP emulators (*DiscGPBasedEmulator*) for more details. First we have a GP or a t-process conditional on some hyperparameters, and second we provide a sample of values of those hyperparameters that are representative of the uncertainty concerning them. See for example the procedure for building a GP emulator for the core problem (*ProcBuildCoreGP*).

Often, this "sample" contains only a single set of values for the hyperparameters, and in this case the emulator is a simple GP or t-process.

The MUCM technology is particularly valuable when the simulator is sufficiently complex that it takes appreciable amounts of computer time to complete just one run, i.e. to evaluate the simulator outputs for a single configuration of input values. The purpose of building the emulator is to facilitate various tasks associated with using the simulator that would be impractical to do directly with the simulator itself because they would require infeasible amounts of computation. Procedures for building an emulator and for using it to carry out various common tasks are presented in this toolkit. For instance, fully Bayesian emulation for the *core problem* is fully documented in the core thread *ThreadCoreGP*.

Some nonstandard tasks may be addressed by a Monte Carlo process of generating sample realisations of the emulator. The emulator is a complete (posterior) probability distribution for the simulator output function $f(\cdot)$. Each sample realisation is an independent random draw from this probability distribution, and so is itself a function. A sample of R realisations $f^{(1)}(\cdot), f^{(2)}(\cdot), \ldots, f^{(R)}(\cdot)$ therefore describes the emulator uncertainty about the simulator output function $f(\cdot)$.

We present here a procedure for generating such a sample of emulator realisations.

8.57.2 Inputs

- An emulator, formulated as a GP or t-process conditional on hyperparameters, plus s sets of hyperparameter values.
- Number of realisations required, R.

8.57.3 Outputs

- Realisations $f^{(k)}(\cdot), \quad k = 1, 2, \ldots, R$.

8.57.4 Procedure

Each realisation begins by selecting one of the sets of hyperparameter values. If $R < s$, we can simply take a random set of values for each realisation or else use a more systematic sample (such as taking only even-numbered hyperparameter sets, if $R = s/2$). If $R > s$ some sets will be reused (and if $s = 1$, i.e. we have only a single set of hyperparameter values, then this set is used for every realisation). This general approach is presented in more detail in the discussion page on Monte Carlo estimation (*DiscMonteCarlo*), where in particular the possibility of obtaining a larger sample of hyperparameter sets is considered.

For the k-th realisation, $f^{(k)}(\cdot)$ is generated by a process that uses a *realisation design* comprising a set of n' points $x'_1, x'_2, \ldots, x'_{n'}$. The discussion page on design for generating emulator realisations (*DiscRealisationDesign*) considers the choice of these points.

Here is the procedure for the k-th realisation:

1. Select a set of hyperparameter values as discussed above.
2. Draw a single random set of predicted values for the outputs at the realisation design points, using the procedure given in *ProcOutputSample*.
3. Rebuild the emulator mean function using these as additional training data. That is, we use the given set of hyperparameters, but the training data design is augmented with the realisation design points, and the training data observation vector is augmented with the sampled predictions obtained in the preceding step.
4. This rebuilt emulator mean function is then $f^{(k)}(\cdot)$.

8.57.5 Additional Comments

The realisation design needs to have enough points so that the variance of the rebuilt emulator is very small at all points of interest; see *DiscRealisationDesign*. If this is not practical, then the sample realisations will not fully account for all the uncertainty in the emulator.

8.58 Procedure: Generating a Sobol's Sequence

8.58.1 Description and Background

The numbers in a Sobol's sequence are generated directly as binary fractions of length w bits. These numbers are created from a special set of w binary fractions V_i called direction numbers. In Sobol's original method, the j-th number X_j is generated by the operation XOR among those direction numbers V_i such that the i-th bit of j is not zero, see comments below. See *DiscSobol* for additional discussion.

8.58.2 Detailed description of the procedure

The i-th number in Sobol's sequence is

$$x_i = \bigoplus_{j=1}^{\infty} b_j(i) v_j,$$

where $b_j(i)$ is the j-th digit of the binary expansion of i; v_j is the j-th direction number and $\oplus$ is the binary (bitwise) XOR operation. The number $b_j(i)$ is a bit and thus it only takes values 0 or 1, i.e. for an integer i we have its binary expansion

$$i = \sum_{j=1}^{\infty} b_j(i) 2^{j-1}.$$

As for any finite number i, its binary expansion has a finite number of non-zero digits, the Sobol's number is computed with a XOR sum of a finite number of direction numbers and thus no convergence properties are needed in the above equations.

An important element in Sobol's sequence are the direction numbers. The direction numbers are binary (dyadic) fractions $v_j \in (0, 1)$ and the resulting Sobol's number is also a binary fraction $x_i \in (0, 1)$. The direction numbers are represented as

$$v_j = \frac{m_j}{2^j}$$

where m_j takes integer values, $1 < m_j < 2^j$, i.e. the direction number v_j is a shift of the binary digits in m_j. The direction numbers are computed by the recursive relation

$$m_j = \left(\bigoplus_{k=1}^{d} 2^k a_k m_{j-k} \right) \bigoplus m_{j-d},$$

where the initial values $m_1, \dots, m_d$ are odd integers and the numbers $a_0, a_1, \dots, a_d$ are given by the coefficients of $p_d(x)$, a primitive polynomial of degree d in Z_2, which is written as

$$p_d(x) = \sum_{k=0}^{d} a_k x^{d-k}.$$

Recall that a primitive polynomial is a polynomial that cannot be represented as the product of two polynomials of lower degree. Primitive polynomials are available in tables in the literature, see Press et al. (1994). For a primitive polynomial, the numbers a_0 and a_d take the value one by definition, and a_0 is not used in the computations.

8.58.3 Running example

In what follows, the subindex $_\mathbf{2}$ refers to binary numbers, everywhere else usual representation (base ten) is assumed.

Take the primitive polynomial $p_3(x) = x^3 + x + 1$, and thus $d = 3$ and $(a_0, a_1, a_2, a_3) = (1, 0, 1, 1)$. The recurrence relation for the direction numbers is

$$\begin{aligned} m_j &= 2a_1 m_{j-1} \oplus 4a_2 m_{j-2} \oplus 8a_3 m_{j-3} \oplus m_{j-3} \\ &= 4m_{j-2} \oplus 8m_{j-3} \oplus m_{j-3} \end{aligned}$$

Using a different primitive polynomial will give a different recurrence relation. To iterate the recurrence relation, initial odd values m_1, m_2, m_3 are needed. Take $m_1 = 1, m_2 = 3, m_3 = 7$.

Now we compute a few extra numbers m_j. We have

$$\begin{aligned} m_4 &= 4(3) \oplus 8(1) \oplus 1 = 12 \oplus 8 \oplus 1 \\ &= 1100_\mathbf{2} \oplus 1000_\mathbf{2} \oplus 1_\mathbf{2} = 101_\mathbf{2} = 5 \end{aligned}$$

and

$$m_5 = 28 \oplus 24 \oplus 3 = 111_\mathbf{2} = 7.$$

The direction numbers are computed by shifting the values obtained, e.g.

$$\begin{aligned} v_1 &= \frac{1}{2} = 0.1_\mathbf{2} \\ v_2 &= \frac{3}{4} = 0.11_\mathbf{2} \\ v_3 &= \frac{7}{8} = 0.111_\mathbf{2} \\ v_4 &= \frac{5}{16} = 0.0101_\mathbf{2} \end{aligned}$$

and

$$v_5 = \frac{7}{32} = 0.00111_\mathbf{2}.$$

The first Sobol's number is obtained with the first direction number

$$x_1 = 1(v_1) = v_1 = 0.1_\mathbf{2} = 0.5.$$

As the binary expansion of two is $10_\mathbf{2}$ then

$$x_2 = 1(v_2) \oplus 0(v_1) = v_2 = 0.11_\mathbf{2} = 0.75$$

and

$$x_3 = v_1 \oplus v_2 = 0.01_\mathbf{2} = 0.25.$$

The following table summarises the construction performed, with the Sobol's sequence on the two rightmost columns, the next to last in binary and the last in base ten.

$i, j(10)$	i, j	$m_j(10)$	m_j	v_j	x_i	$x_i(10)$
1	1	1	1	.1	.1	.5
2	10	3	11	.11	.11	.75
3	11	7	111	.111	.01	.25
4	100	5	101	.0101	.111	.875
5	101	7	111	.00111	.011	.375
6	110			.001	.125	
7	111			.101	.625	
8	1000			.0101	.3125	
9	1001			.1101	.8125	
10	1010			.1001	.5625	

8.58.4 Additional Comments, References and Links

It is important to note that for the first Sobol's number, only the first direction number was needed; then for the following two Sobol's numbers the second direction number was included; for the following four Sobol', the third direction number was included. Without iterating again the recursive relation, 31 Sobol's numbers can be constructed using the first five direction numbers in the above table.

In short, to construct the first $2^k - 1$ Sobol's numbers, we need k direction numbers. If more Sobol's numbers are needed, then the recursive equation must be iterated to obtain direction numbers as required.

Note that by selecting odd initial values $m_1, \dots, m_d$, all the subsequent $m_{d+1}, m_{d+2}, \dots$ are guaranteed to be odd numbers and thus the i-th bit of the direction number v_i is one. This has the important consequence of allowing Sobol's numbers to lie in consecutive finer binary meshes. In other words, a latin hypercube is constructed with the first $2^k - 1$ Sobol's points.

The construction of multivariate Sobol's sequences is achieved by using different primitive polynomials for each dimension. For a table with different primitive polynomials see Press et al. (1994). Sobol's gave a list of recommended primitive polynomials, to avoid high correlations between different dimensions.

An alternative version of Sobol's sequence was due to Antonov and Saleev, who proposed taking instead

$$x_i = \bigoplus_{j=1}^{\infty} g_j(i) v_j$$

where $g_j(i)$ is the j-th digit of the Gray code representation of i. This different Sobol's proposal is faster than the original, as it simplifies the computation to $x_{i+1} = x_i \oplus v_c$, where b_c is the rightmost zero bit in the representation of i.

References:

Antonov, Saleev (1979). USSR Comput. Math. Math. Phys. 19, 252-256.

Bratley, Fox (1988), ACM Trans. Math. Soft. 14(1), 88-100.

Press et al. (1994). Numerical Recipes in C, Cambridge.

8.59 Procedure: Uncertainty Analysis for a Bayes linear emulator

8.59.1 Description and Background

One of the tasks that is required by users of *simulators* is *uncertainty analysis* (UA), which studies the uncertainty in model outputs that is induced by uncertainty in the inputs. Although relatively simple in concept, UA is both important and demanding. It is important because it is the primary way to quantify the uncertainty in predictions made by simulators. It is demanding because in principle it requires us to evaulate the output at all possible values of the uncertain inputs. The *MUCM* approach of first building an *emulator* is a powerful way of making UA feasible for complex and computer-intensive simulators.

This procedure considers the evaluation of the expectaion and variance of the computer model when the input at which it is evaluated is uncertain. The expressions for these quantities when the input x takes the unknown value x_0 are:

$$\mu = \mathrm{E}[f(x_0)] = \mathrm{E}^*[\mu(x_0)]$$

and

$$\Sigma = \mathrm{Var}[f(x_0)] = \mathrm{Var}^*[\mu(x_0)] + \mathrm{E}^*[\Sigma(x_0)]$$

where expectations and variances marked with a * are over the unknown input x_0, and where we use the shorthand expressions $\mu(x) = \mathrm{E}_F[f(x)]$ and $\Sigma(x) = \mathrm{Var}_F[f(x)]$.

8.59.2 Inputs

- An emulator
- The prior emulator beliefs for the emulator in the form of Var$[\beta]$, Var$[w(x)]$
- The training sample design X
- Second-order beliefs about x_0

8.59.3 Outputs

- The expectation and variance of $f(x)$ when x is the unknown x_0

8.59.4 Procedure

Definitions

Define the following quantities used in the procedure of predicting simulator output at a known input (*ProcBLPredict*):

- $\hat{\beta} = \mathrm{E}_F[\beta]$
- $B = \text{Var}[\beta]$
- $\sigma^2 = \text{Var}[w(x)]$
- $V = \text{Var}[f(X)]$
- $e = f(X) - \mathrm{E}[f(X)]$
- $B_F = \text{Var}_F[\beta] = B - BHV^{-1}H^TB^T$

where $f(x)$, β and $w(x)$ are as defined in the thread for Bayes linear emulation of the core model (*ThreadCoreBL*).

Using these definitions, we can write the general adjusted emulator expectation and variance at a **known** input x (as given in *ProcBLPredict*) in the form:

$$\mu(x) = \hat{\beta}^T h(x) + c(x)V^{-1}e$$
$$\Sigma(x) = h(x)^T B_F h(x) + \sigma^2 - c(x)^T V^{-1} c(x) - h(x)^T BHV^{-1}c(x) - c(x)^T V^{-1} H^T Bh(x)$$

where the vector $h(x)$ and the matrix H are as defined in *ProcBLPredict*, $c(x)$ is the $n \times 1$ vector such that $c(x)^T = \text{Cov}[w(x), w(X)]$, and B_F is the adjusted variance of the β.

To obtain the expectation, μ, and variance, Σ, for the simulator at the unknown input x_0 we take expectations and variances of these quantities over x as described below.

Calculating μ

To calculate μ, the expectation of the simulator at the unknown input x_0, we calculate the following expectation:

$$\mu = \mathrm{E}[\mu(x_0)] = \hat{\beta}^T \mathrm{E}[h_0] + \mathrm{E}[c_0]^T V^{-1} d,$$

where we define $h_0 = h(x_0)$ and $c_0^T = \text{Cov}[w(x_0), w(X)]$.

Specification of beliefs for h_0 and c_o is discussed at the end of this page.

Calculating Σ

Σ is defined to be the sum of two components $\mathrm{Var}[\mu(x_0)]$ and $\mathrm{E}^*[\Sigma(x_0)]$. Using h_0 and c_0 as defined above, we can write these expressions as:

$$\begin{aligned}\mathrm{Var}[\mu(x_0)] &= \hat{\beta}^T\mathrm{Var}[h_0]\hat{\beta} + e^T V^{-1}\mathrm{Var}[c_0]^T V^{-1}e + 2\hat{\beta}^T\mathrm{Cov}[h_0, c_0]V^{-1}e \\ \mathrm{E}[\Sigma(x_0)] &= \sigma^2 + \mathrm{E}[h_0]^T B_F \mathrm{E}[h_0] - \mathrm{E}[c_0]^T V^{-1}\mathrm{E}[c_0] - 2\mathrm{E}[h_0]^T BHV^{-1}\mathrm{E}[c_0] \\ &+ \mathrm{tr}\left\{\mathrm{Var}[h_0]B_F - \mathrm{Var}[c_0]V^{-1} - 2\mathrm{Cov}[h_0, c_0]V^{-1}H^T B\right\}\end{aligned}$$

Beliefs about g_0 and c_0

We can see from the expressions given above, that in order to calculate μ and σ, we require statements on the expectations, variances, and covariances for the collection (h_0, c_0). In the Bayes linear framework, it will be straightforward to obtain expectations, variances, and covariances for x_0 however since h_0 and c_0 are complex functions of x_0 it can be difficult to use our beliefs about x_0 to directly obtain beliefs about h_0 or c_0.

In general, we rely on the following strategies:

- Monomial $h(\cdot)$ – When the trend basis functions take the form of simple monomials in x_0, then the expectation, and (co)variance for h_0 can be expressed in terms of higher-order moments of x_0 and so can be found directly. These higher order moments could be specified directly, or found via lower order moments using appropriate assumptions. In some cases, where our basis functions $h(\cdot)$ are not monomials but more complex functions, e.g. $\sin(x)$, these more complex functions may have a particular physical interpretation or relevance to the model under study. In these cases, it can be effective to consider the transformed inputs themselves and thus $h(\cdot)$ becomes a monomial in the transformed space.
- Exploit probability distributions – We construct a range of probability distributions for x_0 which are consistent with our second-order beliefs and our general sources of knowledge about likely values of x_0. We then compute the appropriate integrals over our prior for x_0 to obtain the corresponding second-order moments either analytically or via simulation. When the correlation function is Gaussian, then we can obtain results analytically for certain choices of prior distribution of x_0 – the procedure page on uncertainty analysis using a GP emulator (*ProcUAGP*) addresses this material in detail.

8.60 Procedure: Uncertainty analysis for dynamic emulators

8.60.1 Description and Background

We describe an approximate method for quantifying uncertainty about *dynamic simulator* outputs given uncertainty about the simulator inputs. For a general discussion of uncertainty analysis, see its definition page (*DefUncertaintyAnalysis*) and the procedure page for carrying out uncertainty analysis using a GP emulator (*ProcUAGP*). This method is based on *emulating* the *single step function*, following the procedures described in the variant thread on dynamic emulation (*ThreadVariantDynamic*).

We suppose that there is a true, but uncertain sequence of *forcing inputs* $A_1, \ldots, A_T$ and true values of the simulator parameters ϕ and initial conditions W_0, corresponding to the modelling situation of interest. We denote the true values of all these quantities by a vector X. Uncertainty about X is described by a joint probability distribution $\omega(\cdot)$. The corresponding, uncertain sequence of *state variables* that would be obtained by running the simulator at inputs X is denoted by $W_1, \ldots, W_T$. The procedure we describe quantifies uncertainty about $W_1, \ldots, W_T$ given uncertainty about X.

8.60.2 Inputs

- An emulator for the single step function $w_t = f(w_{t-1}, a_t, \phi)$, formulated as a *GP* or *t-process* conditional on hyperparameters, training inputs D and training outputs $f(D)$.
- A set $\{\theta^{(1)}, \dots, \theta^{(s)}\}$ of emulator hyperparameter values.
- A joint distribution $\omega(\cdot)$ for the forcing variables, initial conditions and simulator parameters

8.60.3 Outputs

- Approximate mean and variance for each of $W_1, \dots, W_T$

8.60.4 Procedure

We describe a Monte Carlo procedure with N defined to be the number of Monte Carlo iterations. For notational convenience, we suppose that $N \le s$. For discussion of the choice of N, including the case $N > s$, see the discussion page on Monte Carlo estimation (*DiscMonteCarlo*).

1. Generate a random value of X from its distribution $\omega(\cdot)$. Denote this random value by X_i
2. Given X_i, and one set of emulator hyperparameters θ_i, iterate the single step emulator using the approximation method described in the procedure page *ProcApproximateIterateSingleStepEmulator* to obtain $\textrm{E}[W_t \| f(D), X_i, \theta^{(i)}]$ and $\textrm{Var}[W_t | f(D), X_i, \theta^{(i)}]$ for all t of interest.
3. Repeat steps 2 and 3 N times and estimate $\textrm{E}[W_t | f(D)]$ and $\textrm{Var}[W_t | f(D)]$ by

$$\hat{\textrm{E}}[W_t | f(D)] = \frac{1}{N} \sum_{i=1}^{N} \textrm{E}[W_t | f(D), X_i, \theta^{(i)}]$$

$$\widehat{\textrm{Var}}[W_t \| f(D)] = \frac{1}{N} \sum_{i=1}^{N} \textrm{Var}[W_t | f(D), X_i, \theta^{(i)}] + \frac{1}{N-1} \sum_{i=1}^{N} \left\{ \textrm{E}[W_t | f(D), X_i, \theta^{(i)}] - \hat{\textrm{E}}[W_t | f(D)] \right\}^2$$

8.60.5 Additional Comments

Note that this procedure does not enable us to fully consider the two sources of uncertainty (uncertainty about inputs and uncertainty about the simulator) separately. (See the discussion page on uncertainty analysis (*DiscUncertaintyAnalysis*)). However, one term that is useful to consider is

$$\frac{1}{N-1} \sum_{i=1}^{N} \left\{ \textrm{E}[w_t | f(D), X_i, \theta^{(i)}] - \hat{\textrm{E}}[W_t | f(D)] \right\}^2 .$$

This gives us the expected reduction in our variance of W_t obtained by learning the true inputs X. If this term is small relative to $\textrm{Var}[W_t | f(D)]$, it suggests that uncertainty about the simulator is large, and that more training runs of the simulator would be beneficial for reducing uncertainty about W_t.

8.61 Procedure: Uncertainty Analysis using a GP emulator

8.61.1 Description and Background

One of the simpler tasks that is required by users of *simulators* is *uncertainty analysis* (UA), which studies the uncertainty in model outputs that is induced by uncertainty in the inputs. Although relatively simple in concept, UA is both important and demanding. It is important because it is the primary way to quantify the uncertainty in predictions made by simulators. It is demanding because in principle it requires us to evaulate the output at all possible values of the uncertain inputs. The *MUCM* approach of first building an *emulator* is a powerful way of making UA feasible for complex and computer-intensive simulators.

This procedure describes how to compute some of the UA measures discussed in the definition page of Uncertainty Analysis (*DefUncertaintyAnalysis*). In particular, we consider the uncertainty mean and variance:

$$\begin{aligned}\mathrm{E}[f(X)] &= \int_{\mathcal{X}} f(x)\omega(x)\mathrm{d}x \\ \mathrm{Var}[f(X)] &= \int_{\mathcal{X}} (f(x) - \mathrm{E}[f(x)])^2\omega(x)\mathrm{d}x\end{aligned}$$

Notice that it is necessary to specify the uncertainty about the inputs through a full probability distribution for the inputs. This clearly demands a good understanding of probability and its use to express personal degrees of belief. However, this specification of uncertainty often also requires interaction with relevant experts whose knowledge is being used to specify values for individual inputs. There is a considerable literature on the elicitation of expert knowledge and uncertainty in probabilistic form, and some references are given at the end of this page.

In practice, we cannot evaluate either $\mathrm{E}[f(X)]$ or $\mathrm{Var}[f(X)]$ directly from the simulator because the integrals require us to know $f(x)$ at every x. Even evaluating numerically by a Monte Carlo integration approach would require a very large number of runs of the simulator, so this is one task for which emulation is very powerful. We build an emulator from a limited *training sample* of simulator runs and then use the emulator to evaluate these quantities. We still cannot evaluate them exactly because of uncertainty in the emulator. We therefore present procedures here for calculating the emulator (i.e. posterior) mean of each quantity as an estimate; while the emulator variance provides a measure of accuracy of that estimate. We use E^* and Var^* to denote emulator mean and variance.

We assume here that a *Gaussian process* (GP) emulator has been built in the form described in the procedure page for building a GP emulator (*ProcBuildCoreGP*), and that we are only emulating a single output. Note that *ProcBuildCoreGP* gives procedures for deriving emulators in a number of different forms, and we consider here only the "linear mean and weak prior" case where the GP has a linear mean function, weak prior information is specified on the hyperparameters β and σ^2 and the emulator is derived with a single point estimate for the hyperparameters δ.

8.61.2 Inputs

- An emulator as defined in *ProcBuildCoreGP*
- A distribution $\omega(\cdots)$ for the uncertain inputs

8.61.3 Outputs

- The expected value $\mathrm{E}^*[\mathrm{E}[f(X)]]$ and variance $\mathrm{Var}^*[\mathrm{E}[f(X)]]$ of the uncertainty distribution mean
- The expected value $\mathrm{E}^*[\mathrm{Var}[f(X)]]$ of the uncertainty distribution variance

8.61.4 Procedure

In this section we describe the calculation of the above quantities. We first give their expressions in terms of a number of integral forms, $U_p, P_p, Q_p, S_p, U, T, R$. We then give the form of these integrals for the *general case* when no assumptions about the form of the distribution of the inputs $\omega(X)$, correlation function $c(X, X')$ and regression function $h(X)$ are made. Finally we give their forms for two special cases.

Calculation of $\mathbf{E}^*[\mathbf{E}[f(X)]]$

$$\mathrm{E}^*[\mathrm{E}[f(X)]] = R\hat{\beta} + Te$$

where

$$e = A^{-1}(f(D) - H\hat{\beta})$$

Calculation of $\mathbf{Var}^*[\mathbf{E}[f(X)]]$

$$\mathrm{Var}^*[\mathrm{E}[f(X)]] = \hat{\sigma}^2[U - TA^{-1}T^{\mathrm{T}} + (R - TA^{-1}H)W(R - TA^{-1}H)^{\mathrm{T}}]$$

where

$$W = (H^{\mathrm{T}}A^{-1}H)^{-1}$$

Calculation of $\mathbf{E}^*[\mathbf{Var}[f(X)]]$

$$\mathrm{E}^*[\mathrm{Var}[f(X)]] = \mathrm{E}^*[\mathrm{E}[f(X)^2]] - \mathrm{E}^*[\mathrm{E}[f(X)]^2]$$

The first term is

$$\begin{aligned}\mathrm{E}^*[\mathrm{E}[f(X)^2]] &= \hat{\sigma}^2[U_p - \mathrm{tr}(A^{-1}P_p) + \mathrm{tr}\{W(Q_p - S_pA^{-1}H - H^{\mathrm{T}}A^{-1}S_p^{\mathrm{T}} + H^{\mathrm{T}}A^{-1}P_pA^{-1}H)\}] \\ &+e^{\mathrm{T}}P_pe + 2\hat{\beta}^{\mathrm{T}}S_pe + \hat{\beta}^{\mathrm{T}}Q_p\hat{\beta}\end{aligned}$$

The second term is

$$\begin{aligned}\mathrm{E}^*[\mathrm{E}[f(X)]^2] &= \hat{\sigma}^2[U - TA^{-1}T^{\mathrm{T}} + \{R - TA^{-1}H\}W\{R - TA^{-1}H\}^{\mathrm{T}}] \\ &+ \left(R\hat{\beta} + Te\right)^2\end{aligned}$$

Dimensions

Before describing the terms involved in the above expressions we first give their dimensions. We assume that we have n *observations*, p *inputs* and q *regression functions*. The dimension of the above quantities are given in the table below.

Symbol	Dimension	Symbol	Dimension
$\hat{\sigma}^2$	1×1	U_p	1×1
$\hat{\beta}$	$q \times 1$	P_p	$n \times n$
e	$n \times 1$	S_p	$q \times n$
f	$n \times 1$	Q_p	$q \times q$
A	$n \times n$	U	1×1
H	$n \times q$	T	$1 \times n$
W	$q \times q$	R	$1 \times q$

The terms $\hat{\sigma}^2$, $\hat{\beta}$, $f(D)$, A and H are defined in *ProcBuildCoreGP*, while e and W are defined above. The terms in the right hand column are inherent in uncertainty analysis and are described below.

The integral forms

General case

When no assumptions are made about the distribution of the inputs, the correlation and the regression functions we have general expressions for the $U_p, P_p, S_p, Q_p, U, R, T$ terms. These are

$$
\begin{aligned}
U_p &= \int_{\mathcal{X}} c(x,x)\omega(x)\mathrm{d}x \\
P_p &= \int_{\mathcal{X}} t(x)t(x)^{\mathrm{T}}\omega(x)\mathrm{d}x \\
S_p &= \int_{\mathcal{X}} h(x)t(x)^{\mathrm{T}}\omega(x)\mathrm{d}x \\
Q_p &= \int_{\mathcal{X}} h(x)h(x)^{\mathrm{T}}\omega(x)\mathrm{d}x
\end{aligned}
$$

$h(x)$ is described in the alternatives page on emulator prior mean function (*AltMeanFunction*). $c(\cdot,\cdot)$ is the correlation function discussed in the alternatives page on emulator prior correlation function (*AltCorrelationFunction*).

Also $t(x) = c(D,x)$, as introduced in *ProcBuildCoreGP*.

Finally, $\omega(x)$ is the joint distribution of the x inputs.

For the U, R, T we have

$$
\begin{aligned}
U &= \int_{\mathcal{X}}\int_{\mathcal{X}} c(x,x')\omega(x)\omega(x')\mathrm{d}x\mathrm{d}x' \\
R &= \int_{\mathcal{X}} h(x)^{\mathrm{T}}\omega(x)\mathrm{d}x \\
T &= \int_{\mathcal{X}} t(x)^{\mathrm{T}}\omega(x)\mathrm{d}x
\end{aligned}
$$

where x and x' are two different realisations of x.

Special case 1

We now show how the above integrals are transformed when we make specific choices about $\omega(\cdot)$ $c(\cdot,\cdot)$ and $h(\cdot)$. We first assume that $\omega(\cdot)$ is a normal distribution given by

$$\omega(x) = \frac{1}{(2\pi)^{d/2}|B|^{-1/2}} \exp\left[-\frac{1}{2}(x-m)^{\mathrm{T}}B(x-m)\right]$$

We also assume the generalised Gaussian correlation function with nugget (see *AltCorrelationFunction*)

$$c(x,x') = \nu I_{x=x'} + (1-\nu)\exp\{-(x-x')^{\mathrm{T}}C(x-x')\}$$

where $I_{x=x'}$ equals 1 if $x = x'$ but is otherwise zero, and ν represents the nugget term. The case of a generalised Gaussian correlation function without nugget is simply obtained by setting $\nu = 0$.

We let both B, C be general positive definite matrices. Also, we do not make any particular assumption for $h(x)$.

We now give the expressions for each of the integrals

$$U_p = 1$$

Note that this result is independent of the existence of a non-zero nugget ν. See the discussion page on the nugget effects in uncertainty and sensitivity (*DiscUANugget*) for more on this point.

P_p is an $n \times n$ matrix, whose $(k,l)^{\rm th}$ entry is

$$P_p(k,l) = (1-\nu)^2 \frac{|B|^{1/2}}{|F|^{1/2}} \exp\left\{-\frac{1}{2}\left[r - g^{\rm T}F^{-1}g\right]\right\}$$

with

$$F = 4C + B$$

and

$$g = 2C(x_k + x_k - 2m)$$

and

$$r = (x_k - m)^{\rm T}2C(x_k - m) + (x_l - m)^{\rm T}2C(x_l - m)$$

The subscripts k and l of x denote training points.

S_p is a $q \times n$ matrix, whose $(k,l)^{\rm th}$ entry is

$$S_p(k,l) = (1-\nu)\frac{|B|^{1/2}}{|F|^{1/2}} \exp\left\{-\frac{1}{2}\left[r - g^{\rm T}F^{-1}g\right]\right\} \mathrm{E}_*[h_k(x)]$$

with

$$F = 2C + B$$

and

$$g = 2C(x_l - m)$$

and

$$r = (x_l - m)^{\rm T}2C(x_l - m)$$

The expectation $\mathrm{E}_*[\cdot]$ is w.r.t. the normal distribution $\mathcal{N}(m + F^{-1}g, F^{-1})$. Also $h_k(x)$ is the k-th element of $h(x)$.

Q_p is a $q \times q$ matrix, whose $(k,l)^{\rm th}$ entry is

$$Q_s(k,l) = \mathrm{E}_*[h_k(x)h_l(x)^{\rm T}]$$

where the expectation $\mathrm{E}_*[\cdot]$ is w.r.t. the normal distribution $\omega(x)$

U is the scalar

$$U = (1-\nu)\frac{|B|}{|F|^{1/2}}$$

with

$$F = \left[\begin{array}{cc} 2C+B & -2C \\ -2C & 2C+B \end{array} \right]$$

R is the $1 \times q$ vector with elements the mean of the elements of $h(x)$, w.r.t. $\omega(x)$, i.e.,

$$R = \int_{\mathcal{X}} h(x)^{\mathrm{T}} \omega(x) \mathrm{d}x$$

T is a $1 \times n$ vector, whose k^{th} entry is

$$T(k) = \frac{(1-\nu)|B|^{1/2}}{|2C+B|^{1/2}} \exp\left\{ -\frac{1}{2} \left[r - g^{\mathrm{T}} F^{-1} g \right] \right\}$$

with

$$F = 2C + B$$

$$g = 2C(x_k - m)$$

$$r = (x_k - m)^{\mathrm{T}} 2C (x_k - m)$$

Special case 2

In this special case, we further assume that the matrices B, C are diagonal. We also consider a special form for the vector $h(x)$, which is the linear function described in *AltMeanFunction*

$$h(x) = [1, x]^{\mathrm{T}}$$

Hence $q = p + 1$. We now present the form of the integrals under the new assumptions.

$$U_p = 1$$

P_p is an $n \times n$ matrix, whose $(k,l)^{\mathrm{th}}$ entry is

$$\begin{array}{rl} P_p(k,l) = & (1-\nu)^2 \prod_{i=1}^{p} \left(\frac{B_{ii}}{4C_{ii}+B_{ii}} \right)^{1/2} \exp\left\{ -\frac{1}{2} \frac{1}{4C_{ii}+B_{ii}} \right. \\ & \left. \left[4C_{ii}^2 (x_{ik} - x_{il})^2 + 2C_{ii}B_{ii} \left\{ (x_{ik} - m_i)^2 + (x_{il} - m_i)^2 \right\} \right] \right\} \end{array}$$

where the double indexed x_{ik} denotes the i^{th} input of the k^{th} training data.

S_p is an $q \times n$ matrix whose $(k,l)^{\mathrm{th}}$ entry is

$$\begin{array}{rl} S_p(k,l) = & (1-\nu) \mathrm{E}_*[h_k(x)] \\ & \prod_{i=1}^{p} \frac{B_{ii}^{1/2}}{(2C_{ii}+B_{ii})^{1/2}} \exp\left\{ -\frac{1}{2} \frac{2C_{ii}B_{ii}}{2C_{ii}+B_{ii}} \left[(x_{il} - m_i)^2 \right] \right\} \end{array}$$

For the expectation we have

$$\begin{array}{ll}\mathrm{E}_*[h_1(x)] = 1 & \\ \mathrm{E}_*[h_{j+1}(x)] = \frac{2C_{jj}x_{jl}+m_jB_{jj}}{2C_{jj}+B_{jj}} & \text{for} \quad j=1,2,\ldots,p\end{array}$$

Q_p is the $q \times q$ matrix,

$$Q_p = \left[\begin{array}{cc} 1 & m^{\mathrm{T}} \\ m & mm^{\mathrm{T}} + B^{-1}\end{array}\right]$$

U is the scalar

$$U = (1-\nu)\prod_{i=1}^{p}\left(\frac{B_{ii}}{B_{ii}+2(2C_{ii})}\right)^{1/2}$$

R is the $1 \times q$ vector

$$R = [1, m^{\mathrm{T}}]$$

T is a $1 \times n$ vector, whose k^{th} entry is

$$T(k) = (1-\nu)\prod_{i=1}^{p}\frac{B_{ii}^{1/2}}{(2C_{ii}+B_{ii})^{1/2}}\exp\left\{-\frac{1}{2}\frac{2C_{ii}B_{ii}}{2C_{ii}+B_{ii}}\left(x_{ik}-m_i\right)^2\right\}$$

8.61.5 References

The topic of eliciting expert judgements about uncertain quantities is dealt with fully in the book

O'Hagan, A., Buck, C. E., Daneshkhah, A., Eiser, J. R., Garthwaite, P. H., Jenkinson, D. J., Oakley, J. E. and Rakow, T. (2006). *Uncertain Judgements: Eliciting Expert Probabilities.* John Wiley and Sons, Chichester. 328pp. ISBN 0-470-02999-4.

For those with limited knowledge of probability theory, we recommend the SHELF package (*disclaimer*), which provides simple templates, software and guidance for carrying out elicitation.

Oakley, J.E., O'Hagan, A., (2002). Bayesian Inference for the Uncertainty Distribution of Computer Model Outputs, *Biometrika*, 89, 4, pp.769-784.

8.61.6 Ongoing work

We intend to provide procedures relaxing the assumption of the "linear mean and weak prior" case of *ProcBuildCoreGP* as part of the ongoing development of the toolkit.

8.62 Procedure: Uncertainty analysis for a function of simulator outputs using multiple independent emulators

8.62.1 Description and Background

Where separate, independent *emulators* have been built for different *simulator* outputs, there is often interest in some function(s) of those outputs. In particular, we may wish to conduct *uncertainty analysis* on a function of the outputs. We assume that, whatever method was used to build each emulator the corresponding toolkit thread also describes how to compute uncertainty analysis for that output alone.

If the function of interest is denoted by $f_0(x)$, then for uncertainty analysis we regard the input vector x as a random variable X having a probability distribution $\omega(x)$ to describe our uncertainty about it. Then uncertainty analysis involves characterising the probability distribution of $f_0(X)$ that is induced by this distribution for X. This distribution is known as the uncertainty distribution. In particular, we are often interested in the uncertainty mean and and variance

$$M_0 = \mathrm{E}[f_0(X)] = \int_{\mathcal{X}} f_0(x)\,\omega(x)\,\mathrm{d}x$$

and

$$V_0 = \mathrm{Var}[f_0(X)] = \int_{\mathcal{X}} (f_0(x) - M_0)^2 \omega(x)\,\mathrm{d}x.$$

The traditional way to compute these quantities is by Monte Carlo methods, drawing many random values of x from its distribution $\omega(x)$ and running the simulator(s) at each sampled input vector to compute the resulting values of $f_0(x)$, which then comprise a sample from the uncertainty distribution. Then for instance M_0 may be estimated by the mean of this sample. The accuracy of this estimate may be quantified using its standard error, which can in principle be made small by taking a very large sample. In practice, this approach is often impractical and in any case the MUCM approach using emulators is generally much more efficient.

The estimate of M_0 is its emulator (posterior) mean, which we denote by $\mathrm{E}^*[M_0]$, while accuracy of this estimate is indicated by the emulator variance $\mathrm{Var}^*[M_0]$. Similarly, the emulator mean of V_0, $\mathrm{E}^*[V_0]$, is the MUCM estimate of V_0.

The individual emulators may be *Gaussian process* (GP) or *Bayes linear* (BL) emulators, although some of the specific procedures given here will only be applicable to GP emulators.

8.62.2 Inputs

- Emulators for r simulator outputs $f_u(x)$, $u = 1, 2, \ldots, r$
- A function $f_0(x)$ of these outputs for which uncertainty analysis is required
- A probability distribution $\omega(.)$ for the uncertain inputs

8.62.3 Outputs

- Estimation of the uncertainty mean M_0, the uncertainty variance V_0 or other features of the uncertainty distribution.

8.62.4 Procedures

Linear case

The simplest case is when the function $f_0(x)$ is linear in the outputs. Thus,

$$f_0(x) = a + \sum_{u=1}^{r} b_u f_u(x),$$

where a and $b_1, b_2, \ldots, b_r$ are known constants. In the linear case, the emulator mean and variance of M_0 may be computed directly from uncertainty means and variances of the individual emulators, and the emulator mean of V_0 requires only a little extra computation.

Let the following be the results of uncertainty analysis of $f_u(X)$, $u = 1, 2, \ldots, r$, where X has the specified distribution $\omega(x)$.

- $\mathrm{E}^*[M_u]$, the emulator mean of the uncertainty mean M_u,
- $\mathrm{Var}^*[M_u]$, the emulator variance of M_u, and
- $\mathrm{E}^*[V_u]$, the emulator mean of the uncertainty variance V_u.

Then

$$\mathrm{E}^*[M_0] = \sum_{u=1}^{r} b_u \mathrm{E}^*[M_u],$$

$$\mathrm{Var}^*[M_0] = \sum_{u=1}^{r} b_u^2 \mathrm{Var}^*[M_u],$$

$$\mathrm{E}^*[V_0] = \sum_{u=1}^{r} b_u^2$$

$$textrmE^*[V_u] + 2\sum_{u<w} (F_{uw} - \mathrm{E}^*[M_u]\mathrm{E}^*[M_w]).$$

The only term in the above formulae that we now need to consider is

$$F_{uw} = \int_{\mathcal{X}} \mathrm{E}^*[f_u(x)]\mathrm{E}^*[f_w(x)]\,\omega(x)\,\mathrm{d}x.$$

For general emulator structures, this can be evaluated very easily and quickly by simulation. We simply draw many random input vectors x from the distribution $\omega(x)$ and in each case evaluate the product of the emulator (posterior) means of the two outputs at the sampled input vector. Given a sufficiently large sample, we can equate F_{uw} to the sample mean of these products. Note that this Monte Carlo computation does not involve running the original simulator(s), and so is typically computationally feasible.

We can do better than this in a special case which arises commonly in practice.

Special case

Suppose that for each $u = 1, 2, \ldots, r$, the emulator of $f_u(x)$ is a GP emulator built using the procedures of the core thread *ThreadCoreGP* and with the following specifications:

1. Linear mean function with basis function vector $h_u(x)$.
2. Weak prior information about the hyperparameters β and σ^2.

Furthermore, suppose that the distribution ω is the (multivariate) normal distribution with mean (vector) m and precision matrix (the inverse of the variance matrix) B.

The emulator will in general include a collection of M sets of values of the correlation hyperparameter matrix B. We present below the computation of F_{uw} for given B, which is therefore the value if $M = 1$. If $M > 1$ the M resulting values should be averaged.

Let $\hat{\beta}_u$, $c_u(x)$ and e_u be the $\hat{\beta}$, $c(x)$ and e vectors for the u-th emulator as defined in the procedure pages for building the GP emulator (*ProcBuildCoreGP*) and carrying out uncertainty analysis (*ProcUAGP*). Then

$$F_{uw} = \hat{\beta}_u^T Q_{uw} \hat{\beta}_w + \hat{\beta}_u^T S_{uw} e_w + \hat{\beta}_w^T S_{wu} e_u + e_u^T P_{uw} e_w,$$

where the matrices Q_{uw}, S_{uw} and P_{uw} are defined as follows:

$$Q_{uw} = \int_{\mathcal{X}} h_u(x)\, h_w(x)^T\, \omega(x)\,\mathrm{d}x,$$

$$S_{uw} = \int_{\mathcal{X}} h_u(x)\, c_w(x)^T\, \omega(x)\,\mathrm{d}x,$$

$$P_{uw} = \int_{\mathcal{X}} c_u(x)\, c_w(x)^T\, \omega(x)\,\mathrm{d}x.$$

Notice that elements of Q_{uw} are just expectations of products of basis functions with respect to the distribution $\omega(x)$, and will usually be trivial to compute in the same way as the matrix Q_p in *ProcUAGP*. Indeed, if all the emulators are built with the same set of basis functions then Q_{uw} is the same for all u, w and equals the Q_p matrix given in *ProcUAGP*.

Similarly, the matrix S_{uw} is the same as the matrix S_p in *ProcUAGP* (for the w-th emulator) except that instead of its own basis function vector we have the vector $h_u(x)$ from the other emulator. If they have the same basis functions, then S_{uw} is just the S_p matrix for emulator w.

Hence it remains only to specify the computation of P_{uw}. This will of course depend on the form of the correlation functions used in building the two emulators.

First suppose that each emulator is built with a generalised Gaussian correlation function (see the alternatives page on emulator prior correlation function (*AltCorrelationFunction*)) which we write for the u-th emulator as

$$\exp\{(x - x')^T D_u (x - x')\}.$$

Then the (k, ℓ) element of F_{uw} is

$$F_{uw}^{k\ell} = |B|^{1/2} |2D_u + 2D_w + B|^{-1/2} \exp(-g/2),$$

where

$$\begin{aligned} g = \quad & 2(m^* - x_k)^T D_u (m^* - x_k) + 2(m^* - x_\ell)^T D_w (m^* - x_\ell) \\ & +(m^* - m)^T B (m^* - m) \end{aligned}$$

and

$$m^* = (2D_u + 2D_w + B)^{-1}(2D_u x_k + 2D_w x_\ell + Bm).$$

Although the generalised Gaussian correlation structure is sometimes used, it is more common to have the simple Gaussian correlation structure in which each D_u is diagonal. If B is also diagonal (so that the various inputs are independent) then the above formulae simplify further.

Simulation-based computation

For GP emulators we have the option of computing uncertainty analysis quantities by simulation-based methods. We generate a large number N of realisations from each of the r emulators, using the approach of the procedure page *ProcSimulationBasedInference*. For each set of realisations, we compute the desired uncertainty analysis property Z;

for instance Z might be M_0, V_0 or the probability that $f_0(X)$ exceeds some threshold. This computation is simply done by Monte Carlo. We then have a sample of values from the posterior distribution of Z, from which for instance we can compute the emulator mean as an estimate of Z and the emulator variance as a summary of emulator uncertainty about Z.

A formal description of this procedure is as follows.

1. For $s = 1, 2, \ldots, N$:
    1. Draw random realisations $f_u^{(s)}(x)$, $s = 1, 2, \ldots, r$, from the emulators
    2. Draw a large sample of random x values from the distribution $\omega(x)$
    3. For each such x, compute $f_0^{(s)}(x)$ from the $f_u^{(s)}(x)$ values
    4. Compute $Z^{(s)}$ from the $f_0^{(s)}(x)$ values
2. From this large sample of $Z^{(s)}$ values, compute the emulator mean and variance, etc.

8.63 Procedure: Recursively update the dynamic emulator mean and variance in the approximation method

8.63.1 Description and Background

This page is concerned with task of *emulating* a *dynamic simulator*, as set out in the variant thread on dynamic emulation (*ThreadVariantDynamic*).

The approximation procedure described in page *ProcApproximateIterateSingleStepEmulator* recursively defines

$$\mu_{t+1} = \mathrm{E}[m^*(w_t, a_{t+1}, \phi)|f(D), \theta]$$

$$V_{t+1} = \mathrm{E}[v^*\{(w_t, a_{t+1}, \phi), (w_t, a_{t+1}, \phi)\}|f(D), \theta] + \mathrm{Var}[m^*(w_t, a_{t+1}, \phi)|f(D), \theta]$$

where the expectations and variances are taken with respect to w_t, with $w_t \sim N_r(\mu_t, V_t)$. Here, we show how to compute μ_{t+1} and V_{t+1} in the case of a *single step* emulator linear mean and a *separable* Gaussian covariance function.

To simplify notation, we now omit the constant parameters ϕ from the simulator inputs. (We can think of ϕ as an extra *forcing input* that is constant over time).

8.63.2 Inputs

- μ_t and V_t
- The single step emulator training inputs D and outputs $f(D)$
- Emulator covariance function parameters B and Σ (defined below).

8.63.3 Outputs

- μ_{t+1} and V_{t+1}

8.63.4 Notation

x: a $p\times 1$ input vector $({x^{(w)}}^T, {x^{(a)}}^T)^T$, with $x^{(w)}$ the corresponding $r\times 1$ vector state variable input, and $x^{(a)}$ the corresponding $(p-r)\times 1$ vector forcing variable input.

$h(x^{(w)}, x^{(a)}) = h(x) = (1\, x^T)^T$: the prior mean function.

$\Sigma c(x,x') = \Sigma \exp\{-(x-x')^T B(x-x')\}$: the prior covariance function, with

Σ: the covariance matrix between outputs at the same input x, and

B: a diagonal matrix with $(i,j)^{\rm th}$ element $1/\delta_i^2$.

B_w: the upper-left $r\times r$ submatrix of :math:‘ B‘.

B_a: the lower-right $(p-r)\times(p-r)$ submatrix of B.

D: the set of training data inputs $x_1,\ldots,n$.

$c\{(x^{(w)}, x^{(a)}), D\} = c(x,D)$: an $n\times 1$ vector with $i^{\rm th}$ element $c(x,x_i)$.

A: an $n\times n$ matrix with $(i,j)^{\rm th}$ element $c(x_i,x_j)$.

H: an $n\times(r+1)$ matrix with $i^{\rm th}$ row $h(x_i)^T$.

$f(D)$: an $n\times r$ matrix of training outputs with $(i,j)^{\rm th}$ element the $j^{\rm th}$ training output type for the $i^{\rm th}$ training input.

$\hat{\beta} = (H^T A^{-1} H)^{-1} H^T A^{-1} f(D)$.

$0_{a\times b}$: an $a\times b$ matrix of zeros.

8.63.5 Procedure

A series of constants need to be evaluated in the following order. Terms defined at each stage can be evaluated independently of each other, but are expressed in terms of constants defined at preceding stages.

Stage 1: compute K_{Vh}, K_{Ec}, K_{Eh}, K_{Ewc} and K_{Ecc}

$$K_{Vh} = \mathrm{Var}[h(w_t,a_{t+1})|f(D),B] = \begin{pmatrix} 0_{1\times 1} & 0_{1\times r} & 0_{1\times(p-r)} \\ 0_{p\times 1} & V_t & 0_{p\times(p-r)} \\ 0_{(p-r)\times 1} & 0_{(p-r)\times r} & 0_{(p-r)\times(p-r)} \end{pmatrix},$$

$K_{Ec} = \mathrm{E}[c\{(w_t,a_{t+1}),D\}|f(D),B]$, an $n\times 1$ vector, with element i given by

$$\begin{aligned} &|2V_tB_w + I_r|^{-1/2}\exp\{-(a_{t+1}-x_i^{(a)})^T B_a(a_{t+1}-x_i^{(a)})\} \\ &\times\exp\{-(\mu_t - x_i^{(w)})^T(2V_t+B_w^{-1})^{-1}(\mu_t - x_i^{(w)})\} \end{aligned}$$

$$K_{Eh} = \mathrm{E}[h(w_t,a_{t+1})|f(D),B] = (1,\mu_t^T,a_{t+1}^T)^T$$

$K_{Ewc} = \mathrm{E}[w_t c\{(w_t, a_{t+1}), D\}^T | f(D), B]$, an $r \times n$ matrix, with column i given by

$$\begin{aligned}\mathrm{E}[w_t c(\{w_t, a_{t+1}\}, x_i) | f(D), B] = |2V_t B_w + I_r|^{-1/2} \times \exp\{-(a_{t+1} - x_i^{(a)})^T B_a (a_{t+1} - x_i^{(a)})\} \\ \times \exp\left\{-(\mu_t - x_i^{(w)})^T \left(2V_t + B_W^{-1}\right)^{-1} (\mu_t - x_i^{(w)})\right\} \times (2B_w + V_t^{-1})^{-1}(2B_w x_i^{(w)} + V_t^{-1}\mu_t).\end{aligned}$$

$K_{Ecc} = \mathrm{E}[c\{(w_t, a_{t+1}), D\} c\{(w_t, a_{t+1}), D\}^T | f(D), B]$, an $n \times n$ matrix, with element i, j given by

$$\begin{aligned}\mathrm{E}[c(\{w_t, a_{t+1}\}, x_i) c(\{w_t, a_{t+1}\}, x_j) | f(D), B] = |4V_t B_w + I_r|^{-1/2} \exp\left\{-\frac{1}{2}(x_i^{(w)} - x_j^{(w)})^T B_w (x_i^{(w)} - x_j^{(w)})\right\} \\ \times \exp\{-(a_{t+1} - x_i^{(a)})^T B_a (a_{t+1} - x_i^{(a)}) - (a_{t+1} - x_j^{(a)})^T B_a (a_{t+1} - x_j^{(a)})\} \\ \times \exp\left[-\left\{\mu_t - \frac{1}{2}(x_i^{(w)} + x_j^{(w)})\right\}^T \left(2V_t + \frac{1}{2}B_W^{-1}\right)^{-1} \left\{\mu_t - \frac{1}{2}(x_i^{(w)} + x_j^{(w)})\right\}\right]\end{aligned}$$

Stage 2: compute K_{Cwc}, K_{Ehh}, and K_{Vc}

$$K_{Cwc} = \mathrm{Cov}[w_t, c\{(w_t, a_{t+1}), D\} | f(D), B] = K_{Ewc} - \mu_t K_{Ec}^T$$

$$K_{Ehh} = \mathrm{E}[h(w_t, a_{t+1}) h(w_t, a_{t+1})^T | f(D), B] = K_{Vh} + K_{Eh} K_{Eh}^T$$

$$K_{Vc} = \mathrm{Var}[c\{(w_t, a_{t+1}), D\} | f(D), B] = K_{Ecc} - K_{Ec} K_{Ec}^T$$

Stage 3: compute K_{Chc}

$$K_{Chc} = \mathrm{Cov}[h(w_t, a_{t+1}), c\{(w_t, a_{t+1}), D\} | f(D), B] = \begin{pmatrix} 0_{1\times n} \\ K_{Cwc} \\ 0_{(p-r)\times n} \end{pmatrix}$$

Stage 4: compute K_{Ehc} and K_{Vm}

$$K_{Ehc} = \mathrm{E}[h(w_t, a_{t+1}) c\{(w_t, a_{t+1}), D\}^T | f(D), B] = K_{Chc} + K_{Eh} K_{Ec}^T$$

$$\begin{aligned}K_{Vm} = \mathrm{Var}[m^*(w_t, a_{t+1}) | f(D), B] = \hat{\beta}^T K_{Vh} \hat{\beta} + \hat{\beta}^T K_{Chc} A^{-1}(f(D) - H\hat{\beta}) \\ +(f(D) - H\hat{\beta})^T K_{Chc}^T \hat{\beta} + (f(D) - H\hat{\beta})^T A^{-1} K_{Vc} A^{-1}(f(D) - H\hat{\beta})\end{aligned}$$

Stage 5: compute K_{Ev}

$$\begin{aligned}K_{Ev} = \mathrm{E}[v^*\{(w_t, a_{t+1}), (w_t, a_{t+1})\} | f(D), B] = 1 - \mathrm{tr}[\{A^{-1} - A^{-1}H(H^T A^{-1} H)^{-1} H^T A^{-1}\} K_{Ecc}] \\ +\mathrm{tr}[(H^T A^{-1} H)^{-1} K_{Ehh}] - 2\mathrm{tr}[A^{-1} H (H^T A^{-1} H)^{-1} K_{Ehc}]\end{aligned}$$

Stage 6: compute the procedure outputs μ_{t+1} and V_{t+1}

$$\mu_{t+1} = K_{Eh}\hat{\beta} + K_{Ec}^T A^{-1}(f(D) - H\hat{\beta})$$

$$V_{t+1} = K_{Vm} + K_{Ev}\Sigma$$

8.63.6 Reference

Conti, S., Gosling, J. P., Oakley, J. E. and O'Hagan, A. (2009). Gaussian process emulation of dynamic computer codes. *Biometrika* 96, 663-676.

8.64 Procedure: Validate a Gaussian process emulator

8.64.1 Description and Background

Once an *emulator* has been built, under the fully *Bayesian Gaussian process* approach, using the procedure in page *ProcBuildCoreGP*, it is important to *validate* it. Validation involves checking whether the predictions that the emulator makes about the *simulator* output accord with actual observation of runs of the simulator. Since the emulator has been built using a *training sample* of runs, it will inevitably predict those correctly. Hence validation uses an additional set of runs, the validation sample.

We describe here the process of setting up a validation sample, using the validation data to test the emulator and interpreting the results of the tests.

We consider here an emulator for the *core problem*, and in particular we are only concerned with one simulator output.

8.64.2 Inputs

- Emulator, as derived in page *ProcBuildCoreGP*.
- The input configurations $D = \{x_1, x_2, \ldots, x_n\}$ at which the simulator was run to produce the training data from which the emulator was built.

8.64.3 Outputs

- A conclusion, either that the emulator is valid or that it is not valid.
- If the emulator is deemed not valid, then indications for how to improve it.

8.64.4 Procedure

The validation sample

The validation sample must be distinct from the training sample that was used to build the emulator. One approach is to reserve part of the training data for validation, and to build the emulator only using the rest of the training data. However, the usual approach to designing a training sample (typically to use points that are well spread out, through some kind of space-filling design, see the alternatives page on training sample design (*AltCoreDesign*)) does not generally provide subsets that are good for validation. It is preferable to develop a validation sample design after building the emulator, taking into account the training sample design D and the estimated values of the correlation function *hyperparameters* δ.

Validation sample design is discussed in page *DiscCoreValidationDesign*. We denote the validation design by $D' = \{x'_1, x'_2, \ldots, x'_{n'}\}$, with n' points. The simulator is run at each of the validation points to produce the output vector $f(D') = (f(x'_1), f(x'_2), \ldots f(x'_{n'})^T)$, where $f(x'_j)$ is the simulator output from the run with input vector x'_j.

We then need to evaluate the emulator's predictions for $f(D')$. For the purposes of our diagnostics, it will be enough to evaluate means, variances and covariances. The procedure for computing these moments is given in the procedure page for predicting simulator outputs (*ProcPredictGP*). The procedure is particularly simple in the case of a linear mean function, weak prior information on hyperparameters β and σ^2, and a single posterior estimate of δ, since then the required moments are simply given by the functions $m^*(\cdot)$ and $v^*(\cdot, \cdot)$ given in *ProcBuildCoreGP* (evaluated at the estimate of δ). In fact, where we have a linear mean function and weak prior information on the other hyperparameters, it is recommended that only a single estimate of δ is computed prior to validation. If the validation tests declare the emulator to be valid, *then* it may be worthwhile to go back and derive a sample of δ values for subsequent use.

We denote the predictive means and covariances of the validation data by $m^*(x'_j)$ and $v^*(x'_j, x'_{j'})$, noting that the predictive variance of the j-th point is $v^*(x'_j, x'_j)$. We let m^* be the mean vector $(m^*(x'_1), m^*(x'_2), \ldots, m^*(x'_{n'}))^T$ and V^* be the covariance matrix with (j, j')-th element $v^*(x'_j, x'_{j'})$.

Possible causes of validation failure

Before presenting the diagnostics it is useful to consider the various ways in which an emulator may fail to make valid predictions. Although the GP is a very flexible way to represent prior knowledge about the computer model, the GP emulator can give poor predictions of simulator outputs for at least two basic reasons. First, the assumption of particular mean and correlation functions may be inappropriate. Second, even if these assumptions are reasonable there are various hyperparameters to be estimated, and a bad or unfortunate choice of training dataset may suggest inappropriate values for these parameters. In the case of the correlation function parameters δ, where we condition on fixed estimates, we may also make a poor choice of estimate.

If the assumed form of the mean function is wrong, for instance because inappropriate regressors have been used in a linear form (see the alternatives page on emulator prior mean function (*AltMeanFunction*)), or if the hyperparameters β have been poorly estimated, then the emulator predictions may be systematically too low or too high in some regions of the input space.

In the various forms of correlation function considered in the discussion page on GP covariance function (*DiscCovarianceFunction*), and in the alternatives page on emulator prior correlation function (*AltCorrelationFunction*) all involve stationarity, implying that we expect the simulator output to respond with similar degrees of smoothness and variability at all points in the input space. In practice, simulators may respond much more rapidly to changes in the inputs at some parts of the space than others. In case of such non-stationarity, credible intervals of emulator predictions can be too wide in regions of low responsiveness or too narrow in regions where the response is more dynamic.

Finally, although the form of the correlation function may be appropriate, we may estimate the parameters σ^2 and δ poorly. When we have incorrect estimation of the variance (σ^2), the credible intervals of the emulator predictions are systematically too wide or too narrow. Poor estimation of the correlation parameters (δ) leads to credible intervals that are too wide or too narrow in the neighbourhood of the training data points.

Validation diagnostics

We present here a basic set of validation diagnostics. In each case we present the diagnostic itself and a reference probability distribution against which the observed value of the diagnostic should be compared. If the observed value is extreme relative to that distribution, i.e. it is far out in one or other tail of the reference distribution, then this indicates a validation failure. It is a matter of judgement how extreme a validation diagnostic needs to be before declaring a validation failure. It is common to use the upper and lower 5% points of the reference distribution as suggestive of a failure, with the upper and lower 0.1% points corresponding to clear evidence of failure.

We discuss the implications and interpretations of each possible validation failure and the extent to which these should lead to a decision that the emulator is not valid.

Reference distributions are approximate, but the approximations are good enough for the purposes of identifying validation failures.

Mahalanobis distance

The Mahalanobis distance diagnostic is

$$M = (f(D') - m^*)^T (V^*)^{-1} (f(D') - m^*).$$

The reference distribution for M is the scaled F-Snedecor distribution with n' and $(n-q)$ degrees of freedom, where q is the dimension of the $h(\cdot)$ function. The mean of this reference distribution is

$$\mathrm{E}[M] = n'$$

and the variance is

$$\mathrm{Var}[M] = \frac{2n'(n'+n-q-2)}{n-q-4}$$

M is a measure of overall fit. If too large it suggests that the emulator is over-confident, in the sense that the uncertainty expressed in V^* is too low compared to the observed differences between the observed $f(D')$ and the predictive means m^*. This in turn may suggest poor estimation of β, under-estimation of σ^2 or generally over-estimated correlation length parameters δ.

Conversely, if M is too small it suggests that the emulator is underconfident, which in turn suggests over-estimation of σ^2 or generally under-estimated correlation length parameters.

An extreme value of this diagnostic should be investigated further through the following more targeted diagnostics. Whilst a moderate value of M generally suggests that the emulator is valid, it is prudent to engage anyway in these further diagnostic checks, because they may bring out areas of concern.

Individual standardised errors

The individual standardised errors are, for $j = 1, 2, \ldots, n'$,

$$e_j = \frac{f(x'_j) - m^*(x'_j)}{\sqrt{v^*(x'_j, x'_j)}}.$$

Each of these is a validation diagnostic in its own right with reference distribution the standard normal distribution, $\mathcal{N}(0,1)$. When comparing with the reference distribution, it is important to remember that we are making many tests and if n' is large enough then we certainly expect some moderately extreme values by pure chance even if the emulator is valid. We are therefore looking for individual very extreme values (larger than 3 in absolute value, say) or patterns of extreme values.

Isolated very extreme e_j values suggest a local irregular behaviour of the simulator in the region of x'_j. Clusters of extreme values whose input values x'_j lie in a particular region of the input space suggest non-stationarity of the simulator in that region.

If large values tend to correspond to x'_j values close to training sample design points this suggests over-estimation of correlation lengths. It should be noted that groups of unusually *small* values of e_j close to training sample design points suggest under-estimation of correlation lengths.

It is important to note, however, that the e_j values are not independent, and this makes interpretation of apparent patterns of individual errors difficult. The next group of diagnostics, the pivoted Cholesky errors, are the most promising of a number of ways to generate independent standardised errors.

Pivoted Cholesky errors

The well-known Cholesky decomposition of a positive-definite matrix yields a kind of square-root matrix. In our diagnostics we use a version of this called the pivoted Cholesky decomposition. The procedure for this is given in page *ProcPivotedCholesky*. Let C be the pivoted Cholesky decomposition of V^* and let

$$t = C^{-1}(f(D') - m^*).$$

Then we consider each of the individual elements t_k of this vector to be a validation diagnostics, for $k = 1, 2, \ldots, n'$. The reference distribution for each t_k is standard normal.

A property of the pivoted Cholesky decomposition is that each t_k is associated with a particular validation sample value, but the ordering of these diagnostics is different from the ordering of the validation dataset. Thus, for instance, the first diagnostic t_1 will not generally correspond to the first validation data point x'_1. The ordering instead assists with identifying particular kinds of emulator failure.

Extreme values of t_k early in the sequence (low k) suggest under-estimation of σ^2, while if the values early in the sequence are unusually small then this suggests over-estimation of σ^2. When these extremes or unusually low values cluster instead at the end of the sequence (high k) it suggests over-/under-estimation of correlation lengths.

Response to diagnostics

If there are no validation failures, or only relatively minor failures, we will generally declare the emulator to be valid. This does not, of course, constitute proof of validity. Subsequent usage may yet uncover problems with the emulator. Nevertheless, we would proceed on the basis that the emulator appears to be valid. In practice, it is rare to have no validation failures - local inhomogeneity of the simulator's behaviour will almost always lead to some emulation difficulties. Declaring validity when minor validation errors have arisen is a pragmatic decision.

When failures cannot be ignored because they are too extreme or too numerous, the emulator should not be used as it stands. Instead, it should be rebuilt with more data. Changes to the assumed mean and correlation functions may also be indicated.

Rebuilding with additional data is, in one sense at least, straightforward since we have the validation sample data which can simply be added to the original training sample data. We now regard the combined data as our training sample and proceed to rebuild the emulator. However, it should be noted that we will need additional validation data with which the validate the rebuilt emulator.

Also, the diagnostics may indicate adding new data in particular regions of the input space, if problems have been noted in those regions. Problems with the correlation parameters may suggest including extra training data points that are relatively close to either the original training data points or the validation points.

8.64.5 Additional Comments

These diagnostics, and some others, were developed in *MUCM* and presented in

Bastos, L. S. and O'Hagan, A. (2008). Diagnostics for Gaussian process emulators. MUCM Technical Report 08/02. (May be downloaded from the MUCM website.)

Validation is something of an evolving art. We hope to extend the discussion here as we gain more experience in *MUCM* with the diagnostics.

8.65 Procedure: Variance Based Sensitivity Analysis using a GP emulator

8.65.1 Description

This page describes the formulae needed for performing *Variance Based Sensitivity Analysis* (VBSA). In VBSA we consider the effect on the output $f(X)$ of a *simulator* as we vary the inputs X, when the variation of those inputs is described by a (joint) probability distribution $\omega(X)$. This probability distribution can be interpreted as describing uncertainty about the best or true values for the inputs, or may simply represent the range of input values of interest to us. The principal measures for quantifying the sensitivity of $f(X)$ to a set of inputs X_w are the main effect, the sensitivity index and the total effect index. We can also define the interaction between two inputs X_i and X_j. These are described below.

Mean, main and interaction effects

Let w denote a subset of the indices from 1 to the number p of inputs. Let X_w denote the set of inputs with indices in w. The mean effect of a set of inputs X_w is the function

$$M_w(x_w) = \mathrm{E}[f(X)|x_w].$$

This is a function of x_w showing how the simulator output for given x_w, when averaged over the uncertain values of all the other inputs, varies with x_w.

The deviation of the mean effect $M_{\{i\}}(x_{\{i\}})$ of the single input X_i from the overall mean,

$$I_i(x_w) = \mathrm{E}[f(X)|x_i] - \mathrm{E}[f(X)].$$

is called the *main effect* of X_i.

Furthermore, we define the *interaction effect* between inputs X_i and X_j to be

$$I_{\{i,j\}}(x_i, x_j) = \mathrm{E}[f(X)|x_i, x_j] - I_i(x_i) - I_j(x_j) - \mathrm{E}[f(X)]$$

Sensitivity variance

The sensitivity variance V_w describes the amount by which the uncertainty in the output is reduced when we are certain about the values of the inputs X_w. That is,

$$V_w = \mathrm{Var}[f(X)] - \mathrm{E}[\mathrm{Var}[f(X)|x_w]]$$

It can also be written in the following equivalent forms

$$V_w = \mathrm{Var}[\mathrm{E}[f(X)|x_w]] = \mathrm{E}[\mathrm{E}[f(X)|x_w]^2] - \mathrm{E}[f(X)]^2$$

Total effect variance

The total effect variance V_{Tw} is the expected amount of uncertainty in the model output that would be left if we removed the uncertainty in all the inputs except for the inputs with indices w. The total effect variance is given by

$$V_{Tw} = \mathrm{E}[\mathrm{Var}[f(X)|x_{\bar{w}}]]$$

where $\bar{w}$ denotes the set of all indices not in w, and hence $x_{\bar{w}}$ means all the inputs except for those in x_w.

V_{Tw} can also be written as

$$V_{Tw} = \text{Var}[f(X)] - V_{\bar{w}}.$$

In order to define the above quantities, it is necessary to specify a full probability distribution for the inputs. This clearly demands a good understanding of probability and its use to express personal degrees of belief. However, this specification of uncertainty often also requires interaction with relevant experts whose knowledge is being used to specify values for individual inputs. There is a considerable literature on the elicitation of expert knowledge and uncertainty in probabilistic form, and some references are given at the end of the procedure page for uncertainty analysis (*ProcUAGP*) page.

We assume here that a *Gaussian process* (GP) emulator has been built according to the procedure page *ProcBuildCoreGP*, and that we are only emulating a single output. Note that *ProcBuildCoreGP* gives procedures for deriving emulators in a number of different forms, and we consider here only the "linear mean and weak prior" case where the GP has a linear mean function, weak prior information is specified on the hyperparameters β and σ^2 and the emulator is derived with a single point estimate for the hyperparameters δ.

The procedure here computes the expected values (with respect to the emulator) of the above quantities.

8.65.2 Inputs

- An emulator as defined in *ProcBuildCoreGP*
- A distribution $\omega(\cdot)$ for the uncertain inputs
- A set w of indices for inputs whose average effect or sensitivity indices are to be computed, or a pair {i,j} of indices defining an interaction effect to be computed
- Values x_w for the inputs X_w, or similarly for X_i, X_j

8.65.3 Outputs

- $\text{E}^*[M_w(x_w)]$
- $\text{E}^*[I_{\{i,j\}}(x_i, x_j)]$
- $\text{E}^*[V_w]$
- $\text{E}^*[V_{Tw}]$

where $\text{E}^*[\cdot]$ denotes an expectation taken with respect to the emulator uncertainty, i.e. a posterior mean.

8.65.4 Procedure

In this section we provide the formulae for the calculation of the posterior means of $M_w(x_w)$, $I_{\{i,j\}}(x_i, x_j)$, V_w and V_{Tw}. These are given as a function of a number of integral forms, which are denoted as U_w, P_w, S_w, Q_w, R_w and T_w. The exact expressions for these forms depend on the distribution of the inputs $\omega(\cdot)$, the correlation function $c(.,.)$ and the regression function $h(\cdot)$. In the following section, we give expressions for the above integral forms for the general and two special cases.

Calculation of $\text{E}^*[M_w(x_w)]$

$$\text{E}^*[M_w(x_w)] = R_w\hat{\beta} + T_w e,$$

where $e = A^{-1}(f(D) - H\hat{\beta})$ and $\hat{\beta}$, A, $f(D)$ and H are defined in *ProcBuildCoreGP*.

For the main effect of X_i the posterior mean is

$$\mathrm{E}^*[I_i(x_i)] = \{R_{\{i\}} - R\}\hat{\beta} + \{T_{\{i\}} - T\}e.$$

It is important to note here that both R_w and T_w are functions of x_w. The dependence on x_w has been suppressed here for notational simplicity.

Calculation of $\mathrm{E}^*[I_{\{i,j\}}(x_i, x_j)]$

$$\begin{aligned}\mathrm{E}^*[I_{\{i,j\}}(x_i, x_j)] = & \quad \{R_{\{i,j\}} - R_{\{i\}} - R_{\{j\}} - R\}\hat{\beta} \\ + & \quad \{T_{\{i,j\}} - T_{\{i\}} - T_{\{j\}} - T\}e\end{aligned}$$

where $R_{\{i,j\}}$ and $R_{\{i\}}$, for instance, are special cases of R_w when the set w of indices comprises the two elements $w = \{i, j\}$ or the single element $w = \{i\}$. Remember also that these will be functions of $x_{\{i,j\}} = (x_i, x_j)$ and $x_{\{i\}} = x_i$ respectively.

Calculation of $\mathrm{E}^*[V_w]$

We write the posterior mean of V_w as

$$\mathrm{E}^*[V_w] = \mathrm{E}^*[\mathrm{E}[\mathrm{E}[f(X)|x_w]^2]] - \mathrm{E}^*[\mathrm{E}[f(X)]^2]$$

The first term is

$$\begin{aligned}\mathrm{E}^*[\mathrm{E}[\mathrm{E}[f(X)|x_w]^2]] \quad &= \hat{\sigma}^2[U_w - \mathrm{tr}(A^{-1}P_w) + \mathrm{tr}\{W(Q_w - S_w A^{-1} H \\ &\qquad -H^{\mathrm{T}} A^{-1} S_w^{\mathrm{T}} + H^{\mathrm{T}} A^{-1} P_w A^{-1} H)\}] \\ &\quad + e^{\mathrm{T}} P_w e + 2\hat{\beta}^{\mathrm{T}} S_w e + \hat{\beta}^{\mathrm{T}} Q_w \hat{\beta}\end{aligned}$$

where $\hat{\sigma}^2$ is defined in *ProcBuildCoreGP*.

The second term is

$$\begin{aligned}\mathrm{E}^*[\mathrm{E}[f(X)]^2] \quad &= \hat{\sigma}^2[U - TA^{-1}T^{\mathrm{T}} + \{R - TA^{-1}H\}W\{R - TA^{-1}H\}^{\mathrm{T}}] \\ &\quad + \left(R\hat{\beta} + Te\right)^2\end{aligned}$$

with $W = (H^{\mathrm{T}} A^{-1} H)^{-1}$.

Calculation of $\mathrm{E}^*[V_{Tw}]$

$\mathrm{E}^*[V_{Tw}]$ can be calculated via the sensitivity variance $V_{\bar{w}}$ using the relation

$$\mathrm{E}^*[V_{Tw}] = \mathrm{E}^*[\mathrm{Var}[f(X)]] - \mathrm{E}^*[V_{\bar{w}}]$$

with

$$\mathrm{E}^*[\mathrm{Var}[f(X)]] = \hat{\sigma}^2[U - TA^{-1}T^{\mathrm{T}} + \{R - TA^{-1}H\}W\{R - TA^{-1}H\}^{\mathrm{T}}]$$

Dimensions

Before presenting the integral forms that appear in the above expressions, we give the dimensions of all the involved quantities in the table below. We assume that we have n *observations*, p *inputs* and q *regression functions*. The terms in the left column are either described in *ProcBuildCoreGP* or they are shorthands (e, W). The terms in the right hand side column are the integral forms, which will be presented in the following section.

Symbol	Dimension	Symbol	Dimension
$\hat{\sigma}^2$	1×1	U_w	1×1
$\hat{\beta}$	$q \times 1$	P_w	$n \times n$
f	$n \times 1$	S_w	$q \times n$
H	$n \times q$	Q_w	$q \times q$
A	$n \times n$	R_w	$1 \times q$
e	$n \times 1$	T_w	$1 \times n$
W	$q \times q$		

The integral forms

General case

When no assumptions are made about the distribution of the inputs, the correlation and the regression functions we have general expressions for the $U_w, P_w, S_w, Q_w, R_w, T_w$ terms. These are

$$U_w = \int_{\mathcal{X}_w}\int_{\mathcal{X}_{\bar{w}}}\int_{\mathcal{X}_{\bar{w}}} c(x,x^*)\omega(x_{\bar{w}}|x_w)\omega(x'_{\bar{w}}|x_w)\omega(x_w)\mathrm{d}x_{\bar{w}}\mathrm{d}x'_{\bar{w}}\mathrm{d}x_w$$

$$P_w = \int_{\mathcal{X}_w}\int_{\mathcal{X}_{\bar{w}}}\int_{\mathcal{X}_{\bar{w}}} t(x)t(x^*)^{\mathrm{T}}\omega(x_{\bar{w}}|x_w)\omega(x'_{\bar{w}}|x_w)\omega(x_w)\mathrm{d}x_{\bar{w}}\mathrm{d}x'_{\bar{w}}\mathrm{d}x_w$$

$$S_w = \int_{\mathcal{X}_w}\int_{\mathcal{X}_{\bar{w}}}\int_{\mathcal{X}_{\bar{w}}} h(x)t(x^*)^{\mathrm{T}}\omega(x_{\bar{w}}|x_w)\omega(x'_{\bar{w}}|x_w)\omega(x_w)\mathrm{d}x_{\bar{w}}\mathrm{d}x'_{\bar{w}}\mathrm{d}x_w$$

$$Q_w = \int_{\mathcal{X}_w}\int_{\mathcal{X}_{\bar{w}}}\int_{\mathcal{X}_{\bar{w}}} h(x)h(x^*)^{\mathrm{T}}\omega(x_{\bar{w}}|x_w)\omega(x'_{\bar{w}}|x_w)\omega(x_w)\mathrm{d}x_{\bar{w}}\mathrm{d}x'_{\bar{w}}\mathrm{d}x_w$$

$$R_w = \int_{\mathcal{X}_{\bar{w}}} h(x)^{\mathrm{T}}\omega(x_{\bar{w}}|x_w)\mathrm{d}x_{\bar{w}}$$

$$T_w = \int_{\mathcal{X}_{\bar{w}}} t(x)^{\mathrm{T}}\omega(x_{\bar{w}}|x_w)\mathrm{d}x_{\bar{w}}$$

Here, $x_{\bar{w}}$ and $x'_{\bar{w}}$ denote two different realisations of $x_{\bar{w}}$. x^* is a vector with elements made up of $x'_{\bar{w}}$ and x_w in the same way as x is composed of $x_{\bar{w}}$ and x_w. Remember also that R_w and T_w are functions of x_w.

$h(x)$ is described in the alternatives page on emulator prior mean function (*AltMeanFunction*). $c(.,.)$ is the correlation function discussed in the alternatives page on emulator prior correlation function (*AltCorrelationFunction*). Also $t(x) = c(D, x)$, as introduced in *ProcBuildCoreGP*.

$\omega(x_w)$ is the joint distribution of the x_w inputs and $\omega(x_{\bar{w}}|x_w)$ is the conditional distribution of $x_{\bar{w}}$ when the values of x_w are known.

Finally, when one of the above integral forms appears without a subscript (e.g. U), it is implied that the set w is empty.

Special case 1

We now show derive closed form expressions for the above integrals when we make specific choices about $\omega(\cdot)$ $c(\cdot,\cdot)$ and $h(\cdot)$. We first assume that $\omega(\cdot)$ is a normal distribution given by

$$\omega(x) = \frac{1}{(2\pi)^{d/2}|B|^{-1/2}}\exp\left[-\frac{1}{2}(x-m)^T B(x-m)\right]$$

We also assume the generalised Gaussian correlation function with nugget (see *AltCorrelationFunction*)

$$c(x,x') = \nu I_{x=x'} + (1-\nu)\exp\{-(x-x')^T C(x-x')\}$$

where $I_{x=x'}$ equals 1 if $x = x'$ but is otherwise zero, and ν represents the nugget term. The case of a generalised Gaussian correlation function without nugget is simply obtained by setting $\nu = 0$.

We let both B, C be general positive definite matrices, partitioned as

$$B = \left[\begin{array}{cc} B_{ww} & B_{w\bar{w}} \\ B_{\bar{w}w} & B_{\bar{w}\bar{w}} \end{array}\right],$$
$$C = \left[\begin{array}{cc} C_{ww} & C_{w\bar{w}} \\ C_{\bar{w}w} & C_{\bar{w}\bar{w}} \end{array}\right]$$

Finally, we do not make any particular assumption for $h(x)$.

We now give the expressions for each of the integrals

U_w is the scalar

$$U_w = (1-\nu)\frac{|B|^{1/2}|B_{\bar{w}\bar{w}}|^{1/2}}{|F|^{1/2}}$$

with

$$F = \left[\begin{array}{ccc} B_{ww} + B_{w\bar{w}}B_{\bar{w}\bar{w}}^{-1}B_{\bar{w}w} & B_{w\bar{w}} & B_{w\bar{w}} \\ B_{\bar{w}w} & 2C_{\bar{w}\bar{w}} + B_{\bar{w}\bar{w}} & -2C_{\bar{w}\bar{w}} \\ B_{\bar{w}w} & -2C_{\bar{w}\bar{w}} & 2C_{\bar{w}\bar{w}} + B_{\bar{w}\bar{w}} \end{array}\right]$$

U is the special case when w is the empty set. The exact formula for U is given in *ProcUAGP*.

P_w is an $n \times n$ matrix, whose $(k,l)^{\mathrm{th}}$ entry is

$$P_w(k,l) = (1-\nu)^2 \frac{|B|^{1/2}|B_{\bar{w}\bar{w}}|^{1/2}}{|F|^{1/2}} \exp\left\{-\frac{1}{2}\left[r - g^{\mathrm{T}}F^{-1}g\right]\right\}$$

with

$$F = \left[\begin{array}{ccc} 4C_{ww} + B_{ww} + B_{w\bar{w}}B_{\bar{w}\bar{w}}^{-1}B_{\bar{w}w} & 2C_{w\bar{w}} + B_{w\bar{w}} & 2C_{w\bar{w}} + B_{w\bar{w}} \\ 2C_{\bar{w}w} + B_{\bar{w}w} & 2C_{\bar{w}\bar{w}} + B_{\bar{w}\bar{w}} & 0 \\ 2C_{\bar{w}w} + B_{\bar{w}w} & 0 & 2C_{\bar{w}\bar{w}} + B_{\bar{w}\bar{w}} \end{array}\right]$$

and

$$g = \left[\begin{array}{l} 2C_{ww}(x_{w,k} + x_{w,l} - 2m_w) + 2C_{w\bar{w}}(x_{\bar{w},k} + x_{\bar{w},l} - 2m_{\bar{w}}) \\ 2C_{\bar{w}w}(x_{w,k} - m_w) + 2C_{\bar{w}\bar{w}}(x_{\bar{w},k} - m_{\bar{w}}) \\ 2C_{\bar{w}w}(x_{w,l} - m_w) + 2C_{\bar{w}\bar{w}}(x_{\bar{w},l} - m_{\bar{w}}) \end{array}\right]$$

and

$$r = (x_k - m)^{\mathrm{T}} 2C(x_k - m) + (x_l - m)^{\mathrm{T}} 2C(x_l - m)$$

P is a special case of P_w when w is the empty set, and reduces to

$$P = T^T T$$

S_w is an $q \times n$ matrix, whose $(k,l)^{\mathrm{th}}$ entry is

$$S_w(k,l) = (1-\nu)\frac{|B|^{1/2}|B_{\bar{w}\bar{w}}|^{1/2}}{|F|^{1/2}} \exp\left\{-\frac{1}{2}\left[r - g^{\mathrm{T}}F^{-1}g\right]\right\} \mathrm{E}_*[h_k(x)]$$

with

$$F = \left[\begin{array}{ccc} 2C_{ww} + B_{ww} + B_{w\bar{w}}B_{\bar{w}\bar{w}}^{-1}B_{\bar{w}w} & B_{w\bar{w}} & 2C_{w\bar{w}} + B_{w\bar{w}} \\ B_{\bar{w}w} & B_{\bar{w}\bar{w}} & 0 \\ 2C_{\bar{w}w} + B_{\bar{w}w} & 0 & 2C_{\bar{w}\bar{w}} + B_{\bar{w}\bar{w}} \end{array}\right]$$

and

$$g = \left[\begin{array}{ccc} 2C_{ww} & 0 & 2C_{w\bar{w}} \\ 0 & 0 & 0 \\ 2C_{\bar{w}w} & 0 & 2C_{\bar{w}\bar{w}} \end{array}\right] \left[\begin{array}{c} x_{w,l} - m_w \\ 0 \\ x_{\bar{w},l} - m_{\bar{w}} \end{array}\right]$$

and

$$r = \left[\begin{array}{c} x_{w,l} - m_w \\ 0 \\ x_{\bar{w},l} - m_{\bar{w}} \end{array}\right]^{\mathrm{T}} \left[\begin{array}{ccc} 2C_{ww} & 0 & 2C_{w\bar{w}} \\ 0 & 0 & 0 \\ 2C_{\bar{w}w} & 0 & 2C_{\bar{w}\bar{w}} \end{array}\right] \left[\begin{array}{c} x_{w,l} - m_w \\ 0 \\ x_{\bar{w},l} - m_{\bar{w}} \end{array}\right]$$

The expectation $\mathrm{E}_z[\cdot]$ is w.r.t. the normal distribution $\mathcal{N}(m + F^{-1}g, F^{-1})$. Also $h_k(x)$ is the k-th element of $h(x)$.

S is a special case of S_w when w is the empty set, and reduces to

$S = R^T T$

Q_w is a $q \times q$ matrix, whose $(k,l)^{\mathrm{th}}$ entry is

$$Q_w(k,l) = \frac{|B|^{1/2}|B_{\bar{w}\bar{w}}|^{1/2}}{|F|^{1/2}} \mathrm{E}_*[h_k(x)h_l^{\mathrm{T}}(x)]$$

where the expectation $\mathrm{E}_*[\cdot]$ is w.r.t. the normal distribution $\mathcal{N}([m_w, m_{\bar{w}}]^{\mathrm{T}}, F^{-1})$

with

$$F = \left[\begin{array}{cc} B_{ww} + B_{w\bar{w}}B_{\bar{w}\bar{w}}^{-1}B_{\bar{w}w} & B_{w\bar{w}} \\ B_{\bar{w}w} & B_{\bar{w}\bar{w}} \end{array}\right]$$

Q is a special case of Q_w when w is the empty set, and reduces to

$$Q = R^T R$$

R_w is the $1 \times q$ vector with elements the mean of the elements of $h(x)$, w.r.t. $\omega(x_{\bar{w}}|x_w)$, i.e.,

$$R_w = \int_{\mathcal{X}_{\bar{w}}} h(x)^{\mathrm{T}} \omega(x_{\bar{w}}|x_w) \mathrm{d}x_{\bar{w}}$$

and is a function of x_w. R is a special case of R_w, when w is the empty set. The formula for R is given in *ProcUAGP*.

T_w is an $1 \times n$ vector, whose k^{th} entry is

$$T_w(k) = (1-\nu)\frac{|B_{\bar{w}\bar{w}}|^{1/2}}{|2C_{\bar{w}\bar{w}} + B_{\bar{w}\bar{w}}|^{1/2}} \exp\left\{-\frac{1}{2}\left[F^{'} + r - g^T F^{-1} g\right]\right\}$$

with

$$\begin{aligned} F^{'} &= (x_w - m_w - (F^{-1}g)_w)^T \\ &\quad \left[2C_{ww} + B_{w\bar{w}}B_{\bar{w}\bar{w}}^{-1}B_{\bar{w}w} - (2C_{w\bar{w}} + B_{w\bar{w}})(2C_{\bar{w}\bar{w}} + B_{\bar{w}\bar{w}})^{-1}(2C_{\bar{w}w} + B_{\bar{w}w})\right] \\ &\quad (x_w - m_w - (F^{-1}g)_w) \\ F &= \left[\begin{array}{cc} 2C_{ww} + B_{w\bar{w}}B_{\bar{w}\bar{w}}^{-1}B_{\bar{w}w} & 2C_{w\bar{w}} + B_{w\bar{w}} \\ 2C_{\bar{w}w} + B_{\bar{w}w} & 2C_{\bar{w}\bar{w}} + B_{\bar{w}\bar{w}} \end{array}\right] \\ g &= 2C(x_k - m) \\ r &= (x_k - m)^T 2C(x_k - m) \end{aligned}$$

$(F^{-1}g)_w$ is the part of the $F^{-1}g$ vector that corresponds to the indices w. According to the above formulation, these are the first $\#(w)$ indices, where $\#(w)$ is the number of indices contained in w.

T is a special case of T_w, when w is the empty set. The formula for T is given in *ProcUAGP*.

Special case 2

In this special case, we further assume that the matrices B, C are diagonal. We also consider a special form for the vector $h(x)$, which is the linear function described in *AltMeanFunction*

$$h(x) = [1, x^{\mathrm{T}}]^{\mathrm{T}}$$

We now present the form of the integrals under the new assumptions.

U_w is the scalar

$$U_w = (1-\nu)\prod_{i\in\bar{w}}\left(\frac{B_{ii}}{B_{ii} + 2(2C_{ii})}\right)^{1/2}$$

Again, U is the special case when w is the empty set, and its exact formula is given in *ProcUAGP*.

P_w is an $n \times n$ matrix, whose $(k,l)^{\mathrm{th}}$ entry is

$$\begin{aligned} P_w(k,l) = \quad & (1-\nu)^2 \textstyle\prod_{i\in\bar{w}} \frac{B_{ii}}{2C_{ii}+B_{ii}} \exp\left\{-\frac{1}{2}\frac{2C_{ii}B_{ii}}{2C_{ii}+B_{ii}}\left[(x_{i,k}-m_i)^2 + (x_{i,l}-m_i)^2\right]\right\} \\ & \textstyle\prod_{i\in w}\left(\frac{B_{ii}}{4C_{ii}+B_{ii}}\right)^{1/2} \exp\left\{-\frac{1}{2}\frac{1}{4C_{ii}+B_{ii}}\right. \\ & \left.\left[4C_{ii}^2(x_{i,k}-x_{i,l})^2 + 2C_{ii}B_{ii}\left\{(x_{i,k}-m_i)^2 + (x_{i,l}-m_i)^2\right\}\right]\right\} \end{aligned}$$

where the double indexed $x_{i,k}$ denotes the i^{th} input of the k^{th} training data.

P is a special case of P_w when w is the empty set, and reduces to

$P = T^T T$

S_w is an $q \times n$ matrix whose $(k,l)^{\mathrm{th}}$ entry is

$$\begin{aligned} S_w(k,l) = \quad & (1-\nu)\mathrm{E}_*[h_k(x)] \\ & \textstyle\prod_{i\in\{w,\bar{w}\}} \frac{B_{ii}^{1/2}}{(2C_{ii}+B_{ii})^{1/2}} \exp\left\{-\frac{1}{2}\left[\frac{2C_{ii}B_{ii}}{2C_{ii}+B_{ii}}(x_{i,l}-m_i)^2\right]\right\} \end{aligned}$$

For the expectation we have

$$\begin{array}{ll} \mathrm{E}_*[h_1(x)] = 1 & \\ \mathrm{E}_*[h_{j+1}(x)] = m_j & \text{if} \quad j \in \bar{w} \\ \mathrm{E}_*[h_{j+1}(x)] = \frac{2C_{jj}x_{j,l}+B_{jj}m_j}{2C_{jj}+B_{jj}} & \text{if} \quad j \in w \end{array}$$

S is a special case of S_w when w is the empty set, and reduces to

$S = R^T T$

Q_w is a $q \times q$ matrix. If we assume that its q indices have the labels $[1, \bar{w}, w]$, then,

$$Q_w = \begin{bmatrix} 1 & m_{\bar{w}}^T & m_w^T \\ m_{\bar{w}} & m_{\bar{w}}m_{\bar{w}}^T & m_{\bar{w}}m_w^T \\ m_w & m_w m_{\bar{w}}^T & m_w m_w^T + B_{ww}^{-1} \end{bmatrix}$$

Q is a special case of Q_w, when w is the empty set, and reduces to

$Q = R^T R$

R_w is a $1 \times q$ vector. If we assume that its q indices have the labels $[1, \bar{w}, w]$, then,

$$R_w = [1, m_{\bar{w}}^T, x_w^T]$$

R is a special case of R_w, when w is the empty set. The formula for R is given in *ProcUAGP*.

T_w is an $1 \times n$ vector, whose k^{th} entry is

$$\begin{aligned} T_w(k) = (1-\nu) \prod_{i\in\{\bar{w}\}} \frac{B_{ii}^{1/2}}{(2C_{ii}+B_{ii})^{1/2}} \quad & \exp\left\{-\frac{1}{2}\frac{2C_{ii}B_{ii}}{2C_{ii}+B_{ii}}(x_{i,k}-m_i)^2\right\} \\ & \exp\left\{-\frac{1}{2}(x_w - x_{w,k})^T 2C_{ww}(x_w - x_{w,k})\right\} \end{aligned}$$

Recall that x_w denotes the fixed values for the inputs X_w, upon which the measures M_w, V_w and V_{Tw} are conditioned. On the other hand, $x_{w,k}$ represents the w inputs of the kth design points.

T is a special case of T_w, when w is the empty set. The formula for T is given in *ProcUAGP*.

8.65.5 References

The principal reference for these procedures is

Oakley, J.E., O'Hagan, A., (2004), Probabilistic Sensitivity Analysis of Complex Models: a Bayesian Approach, *J.R. Statist. Soc. B*, 66, Part 3, pp.751-769.

The above paper does not explicitly consider the case of a non-zero nugget. The calculations of $\mathrm{E}^*[V_w]$ and $\mathrm{E}^*[V_{Tw}]$ produce results that are scaled by $(1-\nu)$, and in general $(1-\nu)\sigma^2$ is the maximum variance reduction achievable because the nugget ν represents noise that we cannot learn about by reducing uncertainty about X. See the discussion page on the nugget effects in sensitivity analysis (*DiscUANugget*) for more details on this point.

8.65.6 Ongoing work

We intend to provide procedures relaxing the assumption of the "linear mean and weak prior" case of *ProcBuildCoreGP* as part of the ongoing development of the toolkit. We also intend to provide procedures for computing posterior variances of the various measures.

8.66 Procedure: Variogram estimation of covariance function hyperparameters

8.66.1 Description and Background

Variogram estimation is an empirical procedure used to *estimate the variance and correlation parameters*, (σ^2, δ) in a stochastic process. Variogram estimation has been typically used to assess the degree of spatial dependence in spatial random fields, such as models based on geological structures. Since the general *emulation* methodology shares many similarities with spatial processes, variogram estimation can be applied to the output of complex computer models in order to assess covariance *hyperparameters*.

Since variogram estimation is a numerical optimisation procedure it typically requires a very large number of evaluations, is relatively computationally intensive, and suffers the same problems as other optimisation strategies such as maximum likelihood approaches.

The variogram itself is defined to be the expected squared increment of the output values between input locations x and x':

$$2\gamma(x,x') = \mathrm{E}[|f(x) - f(x')|^2]$$

Technically, the function $\gamma(x,x')$ is referred to as the "semi-variogram" though the two terms are often used interchangeably. For full details of the properties of and the theory behind variograms, see Cressie (1993) or Chiles and Delfiner (1999).

The basis of this procedure is the result that any two points x and x' which are separated by a distance of (approximately) t will have (approximately) the same value for the variogram $\gamma(x,x')$. Given a large sample of observed values, we can identify all point pairs which are approximately separated by a distance t in the input space, and use their difference in observed values to estimate the variogram.

If the stochastic process $f(x)$ has mean zero then the variogram is related to the variance parameter σ^2 and the *correlation function* as follows

$$2\gamma(x,x') = 2\sigma^2(1 - \mathrm{Cor}[f(x), f(x')]).$$

Thus estimation of the variogram function permits the estimation of the collection of covariance hyperparameters.

Typically, the variogram estimation is applied to the emulator residuals which are a weakly stationary stochastic process with zero mean. For each point at which we have evaluated the computer model, we calculate $w(x) = f(x) - \hat{\beta}^T h(x)$, where $\hat{\beta}$ are the updated values for the coefficients of the linear mean function. We then apply the procedure below to obtain estimates for (σ^2, δ).

8.66.2 Inputs

- A vector of n emulator residuals $(w(x_1), w(x_2), \ldots, w(x_n))$
- The n-point design, X
- A form for the correlation function $c(x,x')$
- Starting values for the hyperparameters (σ^2, δ)

8.66.3 Outputs

- Variogram estimates for (σ^2, δ)

8.66.4 Procedure

Collect empirical information

1. For each pair of residuals $(w(x_i), w(x_j))$, calculate the absolute inter-point distance $h_i = ||x_i - x_j||^2$ and the absolute residual difference $e_{ij} = |w(x_i) - w(x_j)|$
2. Thus we obtain the $n(n-1)/2$ vector of inter-point distances $t = (t_k)$, and the vector of absolute residual differences $e = (e_k)$
3. Divide the range of t into N intervals, $\mathcal{I}_a$
4. For each of the N intervals, calculate:
   1. The number of distances within that interval, $n_a = \#\{t_{ij} : t_{ij} \in \mathcal{I}_a\}$,
   2. The average inter-point separation for that interval, $\bar{t}_a = \frac{1}{n_a} \sum_{t_{ij} \in \mathcal{I}_a} t_{ij}$
   3. An empirical variogram estimate – the classical estimator $\hat{g}_a$, or the robust estimator, $\tilde{g}_a$, as follows:

$$\hat{g}_a = \frac{1}{n_a} \sum_{t_{ij} \in \mathcal{I}_a} e_{ij}^2$$

$$\tilde{g}_a = \frac{(\sum_{t_{ij} \in \mathcal{I}_a} e_{ij}^{0.5}/n_a)^4}{(0.457 + 0.494/n_a)}$$

Either of the two variogram estimators can be used in the estimation procedure, however the classical estimator is noted to be sensitive to outliers whereas the robust estimator has been developed to mitigate this.

Fit the variogram model

Given the statistics n_a, t_a, and an empirical variogram estimate, g_a, for each interval, $\mathcal{I}_a$, we now fit the variogram model by weighted least squares. This typically requires extensive numerical optimisation over the space of possible values for (σ^2, δ).

For a given choice of (σ^2, δ), we calculate the theoretical variogram γ_a for each interval $\mathcal{I}_a$ at mean separation $\bar{t}_a$. The theoretical variogram for a Gaussian correlation function with correlation length parameter δ and at inter-point separation $\bar{t}_a$ is given by

$$\gamma_a = \gamma(\bar{t}_a) = \sigma^2(1 - \exp\{-\bar{t}_a^T M \bar{t}_a\}),$$

where M is a diagonal matrix with elements $1/\delta^2$.

Similarly, the theoretical variogram for a Gaussian correlation function in the presence of a nugget term with variance $\alpha\sigma^2$ is

$$\gamma = \gamma(\bar{t}_a) = \sigma^2(1 - \alpha)(1 - \exp\{-\bar{t}_a^T M \bar{t}_a\}) + \alpha\sigma^2.$$

Beginning with the specified starting values for (σ^2, δ), we then numerically minimise the following expression,

$$W = \sum_a 0.5 n_a (g_a/\gamma_a - 1)^2,$$

for (σ^2, δ) over their feasible ranges.

8.66.5 References

1. Cressie, N., 1993, Statistics for spatial data, Wiley Interscience
2. Chiles, J.P., P. Delfiner, 1999, Geostatististics, Modelling Spatial Uncertainty, Wiley-Interscience

8.67 Procedure: Generate a Weyl design

8.67.1 Description and Background

A Weyl design (also known as a Richtmyer design) is one of a number of non-random space-filling designs suitable for defining a set of points in the *simulator* input space for creating a *training sample*.

The n point Weyl design in p dimensions is generated by a generator set $g = (g_1, \ldots, g_p)$ of irrational numbers. See the "Additional Comments" below for discussion of the choice of generators.

8.67.2 Inputs

- Number of dimensions p
- Number of points desired n
- Set of irrational generators $g_1, \ldots, g_p$

8.67.3 Outputs

- Weyl design $D = \{x_1, x_2, \ldots, x_n\}$

8.67.4 Procedure

For $j = 0, \ldots, n-1$, generate points as

$$x_{j+1} = (j \times g_1 \mathrm{mod} 1, j \times g_2 \mathrm{mod} 1, \ldots, j \times g_d \mathrm{mod} 1) \,.$$

Note that the operator "mod 1" here has the effect of returning the fractional part of each number. For instance, if $j = 7$ and $g_1 = \sqrt{2} = 1.414\ldots$, then

$$j \times g_1 = 9.89949\ldots$$

and so

$$j \times g_1 \mathrm{mod} 1 = 0.89949\ldots.$$

8.67.5 Additional Comments

A potential problem with Weyl designs is the difficulty in finding suitable generators. One suggestion is to let g_i be the square root of the i-th prime, but this may not work well when p is large.

8.67.6 References

The following is a link to the repository for Matlab code for the Weyl sequence in up to 100 dimensions: CPWeylSequence.m (*disclaimer*).

8.68 Example: 1 Dimensional History Matching

8.68.1 Description

In this Example page we outline a simple but intuitive 1-dimensional example of the history matching process, as described in *ThreadGenericHistoryMatching*. This will require matching a simple 1-dimensional simulator $f(x)$ to an observation z, and will involve two waves of refocussing. The notation used is the same as that defined in *ThreadGenericHistoryMatching*. Here we use the term model synonymously with the term *simulator*.

8.68.2 Setup

1. We have a simple 1D function (a sine wave), which we have used to perform 6 runs.
2. The plan is that we will construct an emulator, and use this to evaluate the implausibility function $I(x)$ over the 1D input space $\mathcal{X}_0$
3. We will then impose cutoffs to reduce the 1D input space $\mathcal{X}_0$ down to the non-implausible region denoted $\mathcal{X}_1$ (see Wave 1 and Figure 1 below).
4. Then we perform a second wave of runs (that have inputs belonging to $\mathcal{X}_1$) and re-emulate using these new runs.
5. The new wave 2 emulator will be more accurate, and hence will change the implausibility function $I(x)$.
6. Imposing the cutoffs on the new implausibility function defines the non-implausible region $\mathcal{X}_2$, which will be (in this simple example) a good approximation to the set of all acceptable inputs $\mathcal{X}$ (see Wave 2 and Figure 2 below).
7. We do not discuss the emulator construction, except to say here we are using purely a Gaussian process emulator, with no regression terms (this is to allow easier visualisation).

8.68.3 Wave 1

Figure 1 (top panel) shows:

1. The six model runs as black dots (inputs on x-axis, outputs on y-axis).
2. The emulator expectation $\mathrm{E}[f(x)]$ (blue line).
3. Suitable credible intervals (red lines) given by $\mathrm{E}[f(x)] \pm 3\sqrt{\mathrm{Var}}[f(x)]$.
4. The observation $z = -0.8$ along with $\pm 3\sigma$ observational errors, where $\sigma^2 = \mathrm{Var}[e] = 0.05^2$ (all given by the 3 horizontal black lines).

Figure 1 (bottom panel) shows:

1. The 1D implausibility function $I(x)$ (black dots).
2. The implausibility cutoff level $c = 3$ (thin green line).
3. The green colouring on the x axis shows the inputs that belong to the non-implausible region after Wave 1, denoted $\mathcal{X}_1$

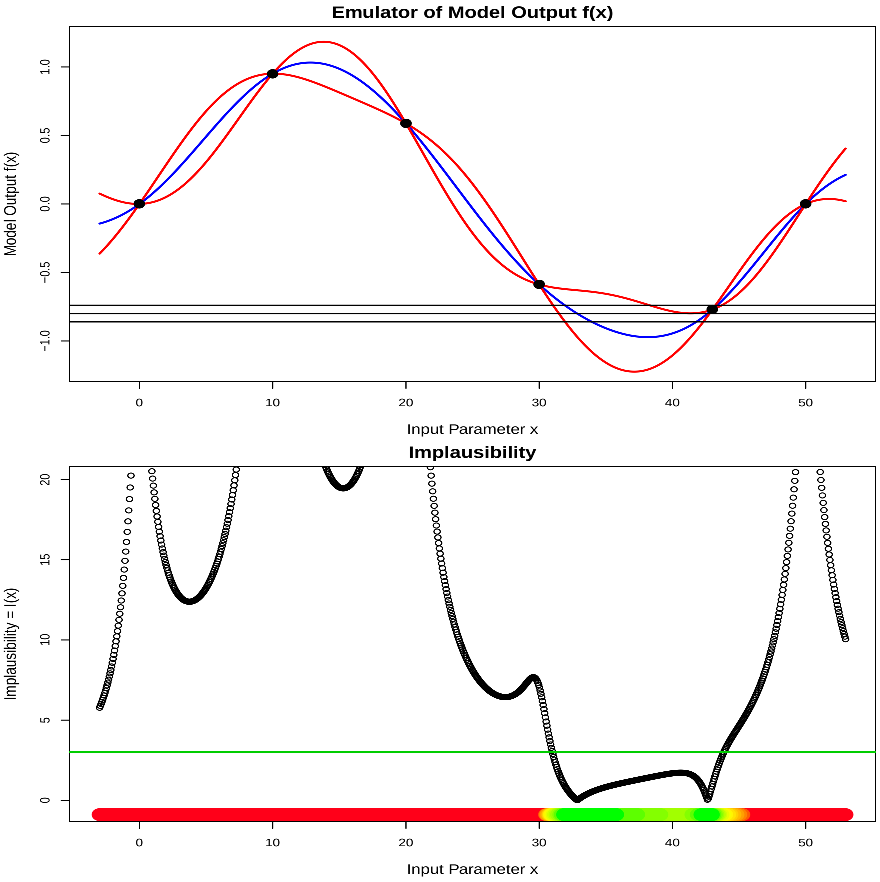


Fig. 3: **Figure 1:** 1D history matching: Wave 1

At this point we perform three more runs within the green region of Figure 1, that is pick new x values that are members of $\mathcal{X}_1$ and run the model at these new points.

8.68.4 Wave 2

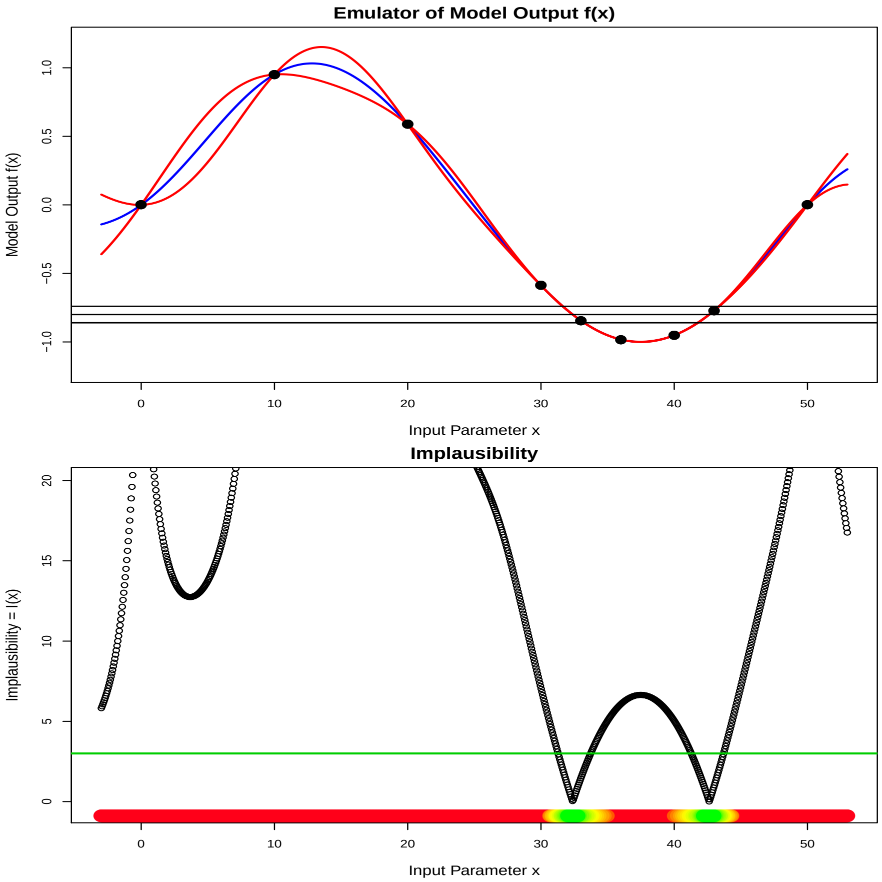


Fig. 4: **Figure 2:** 1D history matching: Wave 2

Figure 2 (top panel) shows how the emulator looks at Wave 2 after the three new runs have been incorporated. Note that:

1. The new runs are only in the previous non-implausible region $\mathcal{X}_1$.
2. The emulator is now far more accurate in this $\mathcal{X}_1$ region (the credible interval given by the red lines is much narrower).
3. Further Waves would not be useful as the emulator variance is now far smaller that the observational errors, hence a Wave 3 would not teach us much more about the set of acceptable inputs $\mathcal{X}$.

Figure 2 (bottom panel) shows the new implausibility measure $I(x)$ at Wave 2. Note that:

1. The implausibility measure $I(x)$ has increased in certain regions (because we have more information from the 3 new runs).
2. The cutoff now defines a smaller non-implausible set $\mathcal{X}_2$ (given by the green points on the x-axis): this is Refocussing.
3. There are now two non-implausible regions of input space remaining: a definite possibility in many applications.

8.68.5 Discussion

The 1-Dimensional example shows the basic history matching process. Note that the model discrepancy was assumed zero for simplicity and to aid visualisation.

When dealing with higher dimensional input spaces, the problem of visualising the results (e.g. the location and shape of the current non-implausible volume $\mathcal{X}_j$) becomes important. Various techniques are available including implausibility projections and optical depth plots. See Vernon et. al 2010 for further details.

8.68.6 References

Vernon, I., Goldstein, M., and Bower, R. (2010), "Galaxy Formation: a Bayesian Uncertainty Analysis," MUCM Technical Report 10/03

8.69 Example: A one dimensional emulator

8.69.1 Model (simulator) description

In this page we present an example of fitting an emulator to a $p = 1$ dimensional simulator (model). The simulator we use is surfebm. Surfebm is an energy balance model of the Earth's climate. The state variables are upper ocean temperatures averaged around the globe at constant latitudes. There are 18 boxes from the North to South poles. In addition there is a single deep ocean box. If the upper ocean boxes have a temperature below 0 it freezes. Water and ice have different albedos and there is an overturning circulation moving heat from the Arctic to the Antarctic. In this example we will keep all inputs fixed, apart from one, the solar constant. For simplicity in this example we only look at a single output, the mean upper ocean temperature. Figure 1 shows the mean temperature over the input range of the solar constant.

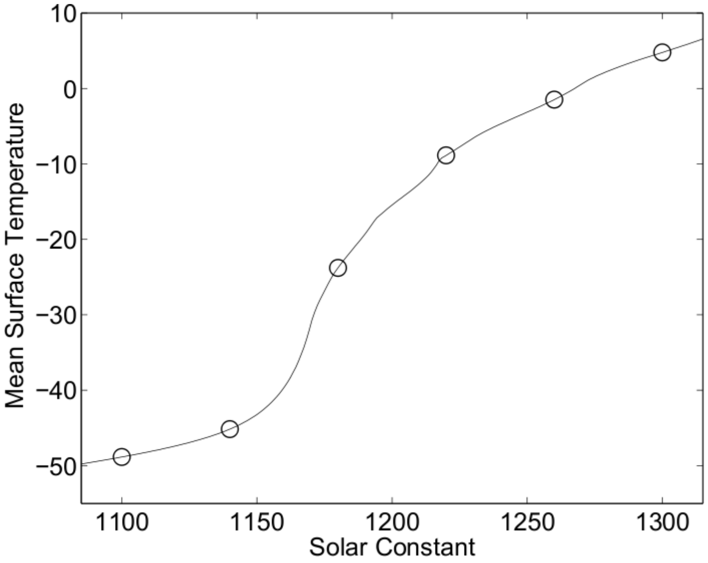


Fig. 5: **Figure 1:** The simulator's output and the 6 design points

8.69.2 Design

Design is the selection of the input points at which the simulator is to be run. There are several design options that can be used, as described in the alternatives page on training sample design for the core problem (*AltCoreDesign*). For this example, we will use an Optimised Latin Hypercube Design, which for one dimension is a set of equidistant points on the space of the input variable.

We select $n = 6$ design points, which are shown in Figure 1 as circles. These points are

$$[\tilde{x}_1, \tilde{x}_2, \cdots, \tilde{x}_6] = [1100, 1140, 1180, 1220, 1260, 1300]$$

We scale our design points so that they lie in $[0, 1]$. That is,

$$D \equiv [x_1, x_2, \cdots, x_6] = [0.0, 0.2, \cdots, 1.0]$$

The output of the model at these points is

$$f(D) = [-48.85, -45.15, -23.78, -8.87, -1.49, 4.77]^T$$

8.69.3 Gaussian Process setup

In setting up the Gaussian process, we need to define the mean and the covariance function. For the mean function, we choose the linear form described in the alternatives page for emulator prior mean function (*AltMeanFunction*), which is $h(x) = [1, x]^T$ and $q = 1 + p = 2$.

For the covariance function we choose $\sigma^2 c(\cdot, \cdot)$, where the correlation function $c(\cdot, \cdot)$ has the Gaussian form described in the alternatives page on emulator prior correlation function (*AltCorrelationFunction*)

$$c(x, x') = \exp\left[-\sum_{i=1}^{p}\{(x_i - x'_i)/\delta_i\}^2\right] = \exp\left[-\{(x - x')/\delta\}^2\right]$$

noting that in this case $p = 1$.

8.69.4 Estimation of the correlation length

We start with the estimation of the correlation length δ. In this example we will use the value of δ that maximises the posterior distribution $\pi^*_\delta(\delta)$, assuming that there is no prior information on δ, i.e. $\pi(\delta) \propto \mathrm{const}$. The expression that needs to be maximised, according to the procedure page on building a Gaussian process emulator for the core problem (*ProcBuildCoreGP*), is

$$\pi^*_\delta(\delta) \propto (\widehat{\sigma}^2)^{-(n-q)/2}|A|^{-1/2}\|H^T A^{-1} H|^{-1/2}.$$

where

$$\widehat{\sigma}^2 = (n - q - 2)^{-1} f(D)^T \left\{A^{-1} - A^{-1}H\left(H^T A^{-1} H\right)^{-1} H^T A^{-1}\right\} f(D).$$

H is defined as

$$H = [h(x_1), h(x_2), \cdots, h(x_n)]^T.$$

A is the $n \times n$ correlation matrix of the design points, with elements $c(x_i, x_j), i, j \in \{1, \cdots, n\}$.

Recall that in the above expressions the only term that is a function of δ is the correlation matrix A.

Reparameterisation

In order to maximise the posterior $\pi^*_\delta(\delta)$, we reparameterise it using $\tau = \ln(2\delta)$, to obtain $\pi^*(\tau) = \pi^*_\delta(\exp(\tau/2))$. This has the benefit of making the optimisation problem unconstrained, because $\delta \in (0, \infty)$, while $\tau \in (-\infty, \infty)$.

Posterior function

Figure 2 shows $\ln(\pi^*(\tau))$ as a function of τ.

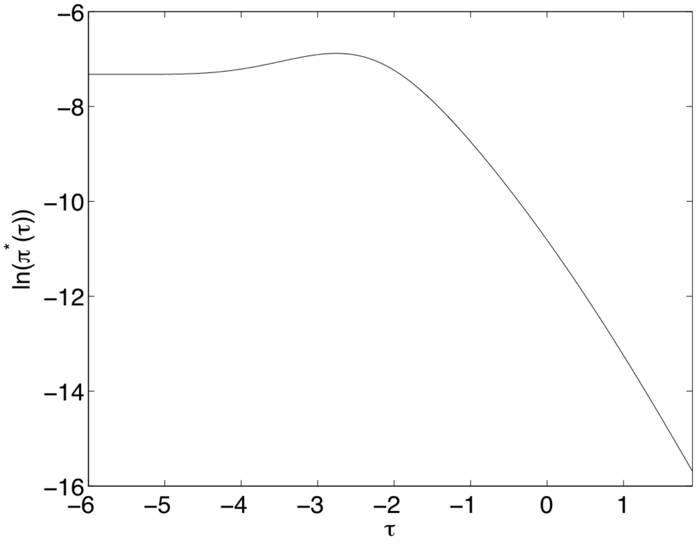


Fig. 6: **Figure 2:** The log posterior of τ as a function of τ

The maximum of this value can be obtained with any maximisation algorithm. In our case, we used Nelder - Mead. The value of τ that maximises the posterior is -2.76 and the respective value of δ is 0.25. From now on, we will refer to this value as $\hat{\delta}$.

Because we have scaled the input to lie in $[0, 1]$, $\hat{\delta}$ is one quarter of the range of x over which we are fitting the emulator. This is a fairly typical value for a smoothness parameter, indicating that there is structure to the way that the output responds to this input in addition to the linear mean function that has been fitted, but that the response is not particularly "wiggly". In terms of the original input scale, $\hat{\delta}$ corresponds to a smoothness parameter of $200 \times 0.25 = 50$.

8.69.5 Estimates for the remaining parameters

Apart from the correlation lengths, the two other parameters of the Gaussian Process are β and σ^2. Having estimated the correlation lengths, the estimate for $\widehat{\sigma}^2$ is given by the equation above and

$$\hat{\beta} = \left(H^T A^{-1} H\right)^{-1} H^T A^{-1} f(D) \,.$$

Note that in these equations, the matrix A is calculated using $\hat{\delta}$. The values we got for the above parameters are $\hat{\beta} = [-47.30, 53.79]^T$ and $\hat{\sigma}^2 = 92.89$. Therefore, the fitted underlying linear trend is $y = -47.3 + 53.79x$ (remembering that x is transformed to lie in [0,1]), with deviation from this line measured by a standard deviation of $\sqrt{92.89} = 9.64$.

8.69.6 Posterior mean and Covariance functions

The expressions for the posterior mean and covariance functions according to *ProcBuildCoreGP* are

$$m^*(x) = h(x)^T\widehat{\beta} + c(x)^T A^{-1}(f(D) - H\widehat{\beta})$$

and

$$v^*(x,x') = \widehat{\sigma}^2\{c(x,x') - c(x)^T A^{-1} c(x') + \left(h(x)^T - c(x)^T A^{-1} H\right)\left(H^T A^{-1} H\right)^{-1}\left(h(x')^T - c(x')^T A^{-1} H\right)^{\mathrm{T}}\}.$$

Figure 3 shows the predictions of the emulator for 100 points uniformly spaced in $1075, 1325$ in the original scale. The continuous line is the output of the simulator and the dashed line is the emulator's output m^*. The shaded areas represent 2 times the standard deviation of the emulator's prediction, which is the square root of the diagonal of matrix v^*.

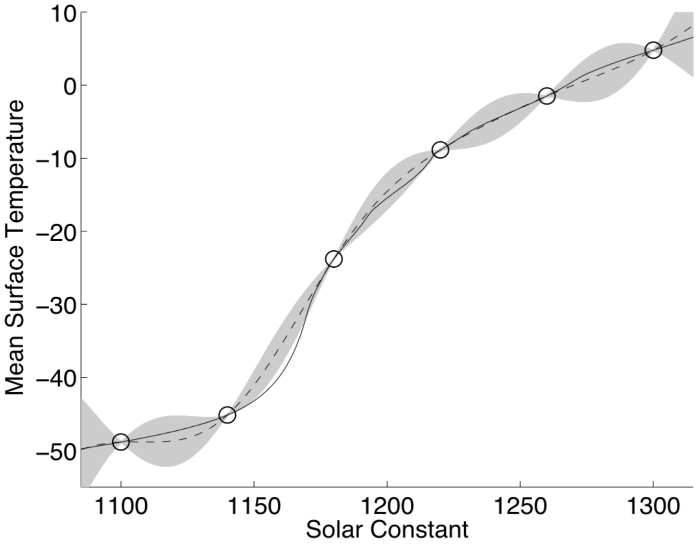


Fig. 7: **Figure 3:** Simulator (continuous line), emulator's mean (dashed line) and 95% confidence intervals (shaded area)

We see that the true simulator output lies outside the two standard deviation interval for inputs around 1160 but otherwise seems to capture the behaviour of the simulator well. The next step would be to validate the emulator, at which point the anomaly around 1160 might be found, leading to a rebuilding and improvement of the emulator.

8.69.7 Validation

In this section we validate the above emulator according to the procedure page on validating a Gaussian process emulator (*ProcValidateCoreGP*).

The first step is to select the validation design. Following the advice in the discussion page on the design of a validation sample (*DiscCoreValidationDesign*), we chose $n' = 3p = 3$. We choose 2 points close to the original training points and one more halfway between 1140 and 1180, where the emulator makes the biggest jump. In the original input space of the simulator, the validation points we choose are

$$[\tilde{x}'_1, \tilde{x}'_2, \tilde{x}'_{n'}] = [1110, 1160, 1250]$$

and in the transformed space

$$D' = [x'_1, x'_2, x'_3] = [0.05, 0.3, 0.75]$$

The simulator's output at these points is

$$f(D') = [-48.16, -39.63, -3.14]^T$$

We then calculate the mean $m^*(\cdot)$ and variance $v^*(\cdot, \cdot)$ of the emulator at the validation design point D'. The means are

$$m^*(D') = [-48.83, -35.67, -3.11]^T$$

and the standard deviations are

$$\text{diag}[v^*(D', D')]^{1/2} = [1.38, 1.34, 0.98]^T$$

The predictions with error bars at 2 standard deviations are shown in Figure 4, which shows that all the predictions are within 2 standard deviations, with the exception of the prediction at $\tilde{x} = 1160$.

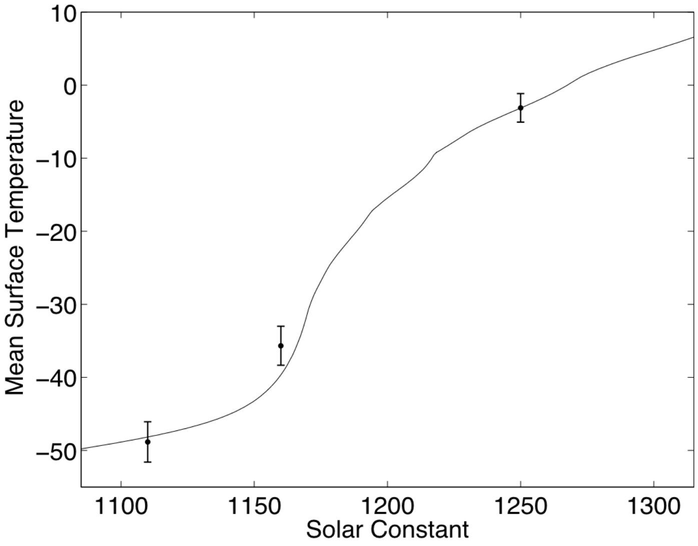


Fig. 8: **Figure 4:** Emulator's predictions at the validation points, with error bars at ±2 standard deviations

Figure 5, shows the individual standardised errors, indicating that they are all within the boundaries of ±2 standard deviations, with the exception of the prediction at 1160, which is at 3 standard deviations.

The Mahalanobis distance diagnostic for this validation was found to be

$$M = (f(D') - m^*(D'))^T (v^*(D', D'))^{-1} (f(D') - m^*(D')) = 26.6$$

when its theoretical mean is

$$\text{E}[M] = n' = 3$$

The conclusion is that the emulator failed in both diagnostics, because the Mahalanobis distance is too large, and one of the individual standardised errors is close to 3. We will therefore rebuild the emulator using both the previous training points and the validation points for training.

Rebuilding the emulator

We now rebuild the emulator using both the training and the validation points. That is,

$$[\tilde{x}_1, \tilde{x}_2, \cdots, \tilde{x}_n,] = [1100, 1110, 1140, 1160, 1180, 1220, 1250, 1260, 1300]$$

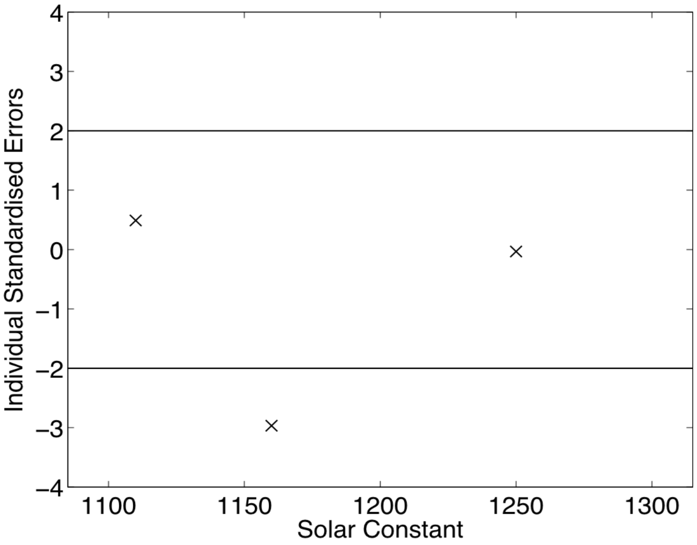


Fig. 9: **Figure 5:** Individual standardised errors for the prediction at the validation points

which in the transformed space are

$$D = [x_1, x_2, \cdots, x_n,] = [0, 0.05, 0.2, 0.3, 0.4, 0.6, 0.75, 0.8, 1]$$

The simulator's output at these points is

$$f(D) = [-48.85, -48.16, -45.15, -39.63, -23.78, -8.87, -3.14, -1.48, 4.77]^T$$

Following the same procedure as before, we find the following values for the parameters:

$$\begin{aligned}\hat{\delta} &= 0.17\\ \hat{\sigma}^2 &= 50.00\\ \hat{\beta} &= [-48.65, 56.47]^T\end{aligned}$$

Figure 6 shows the predictions of the emulator for 100 points uniformly spaced in 1075, 1325 in the original scale. A comparison with Figure 3 shows that the new emulator has a smaller predictive variance, and the mean of the prediction is closer to the actual value of the simulator.

Re-validation

We finally carry out the two validation tests for the new emulator. Using the same reasoning as before, we select the following validation points

$$[\tilde{x}'_1, \tilde{x}'_2, \tilde{x}'_{n'}] = [1130, 1200, 1270]$$

which in the transformed input space are

$$D' = [x'_1, x'_2, x'_3] = [0.15, 0.5, 0.85]$$

The simulator's output at these points is

$$f(D') = [-48.16, -39.63, -3.14]^T$$

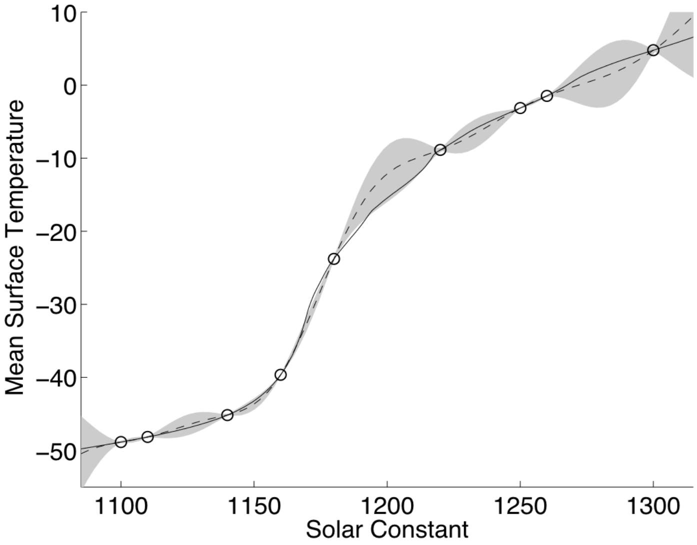


Fig. 10: **Figure 6:** Mean and 95% confidence intervals for the rebuilt emulator

The following the same new validation points we select are

$$D' = [x'_1, x'_2, \cdots, x'_{n'}] = [0.05, 0.15, \cdots, 0.95]$$

The simulator's output for these values is

$$f(D') = [-46.42, -15.45, 0.55]^T$$

The mean and standard deviation of the emulator at the new validation points is

$$m^*(D') = [-45.98, -12.15, -0.23]^T$$

$$\text{diag}[v^*(D', D')]^{1/2} = [0.60, 1.90, 0.99]^T$$

Figure 7 shows the individual standardised errors. Note that they are now all within 2 standard deviations.

The Mahalanobis distance is now $M = 6.17$, whereas its theoretical mean is

$$\text{E}[M] = n' = 3.$$

and its variance is

$$\text{Var}[M] = \frac{2n'(n' + n - q - 2)'}{n - q - 4} = 16.$$

Therefore, the Mahalanobis distance found is a typical value from its reference distribution, and taking also into account the fact that all the standardised errors are within 2 standard deviations, the emulator is declared valid.

8.69.8 Final Build

After the validation being successful, we can rebuild the emulator using all the data, which would provide us with its final version. The final set of 12 training points is

$$[\tilde{x}_1, \tilde{x}_2, \cdots, \tilde{x}_{n},] = [1100, 1110, 1130, 1140, 1160, 1180, 1200, 1220, 1250, 1260, 1270, 1300]$$

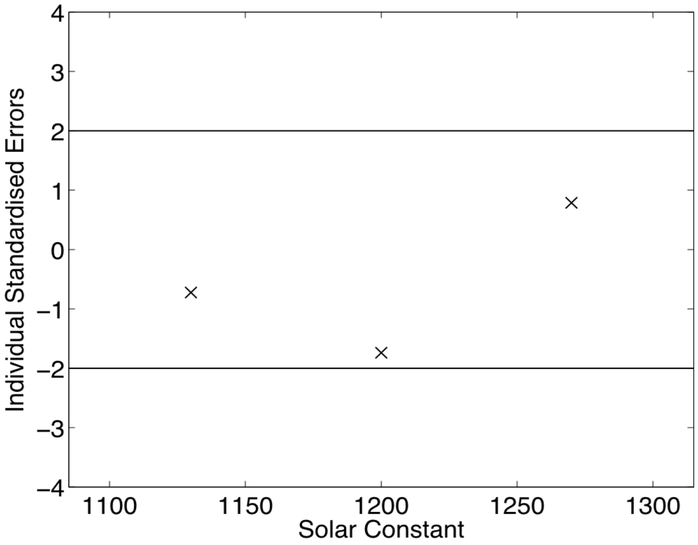


Fig. 11: **Figure 7:* individual standardised errors for the new validation points

which in the transformed space are

$$D = [x_1, x_2, \cdots, x_n,] = [0, 0.05, 0.15, 0.2, 0.3, 0.4, 0.5, 0.6, 0.75, 0.8, 0.85, 1]$$

The simulator's output at these points is

$$f(D) = [-48.85, -48.16, -46.42, -45.15, -39.63, -23.78, -15.45, -8.87, -3.14, -1.49, 0.55, 4.77]^T$$

The hyperparameter estimates obtained using the above training points are

$$\begin{aligned} \hat{\delta} &= 0.145 \\ \hat{\sigma}^2 &= 30.05 \\ \hat{\beta} &= [-50.18, 58.64]^T \end{aligned}$$

Finally, figure 8 shows the predictions of the emulator for 100 points, uniformly spaced in 1075, 1325 in the original scale. Comparing with figures 3 and 6, we see that the addition of the extra training points has further reduced the predictive variance and moved the mean closer to the simulator's output.

8.70 Example: A two dimensional emulator with uncertainty and sensitivity analysis

8.70.1 Model (simulator)

In this example we fit a Gaussian process emulator to the surfebm model, using two inputs, the Solar Constant and the Albedo, and one output, the Mean Surface Temperature.

8.70.2 Fitting the emulator

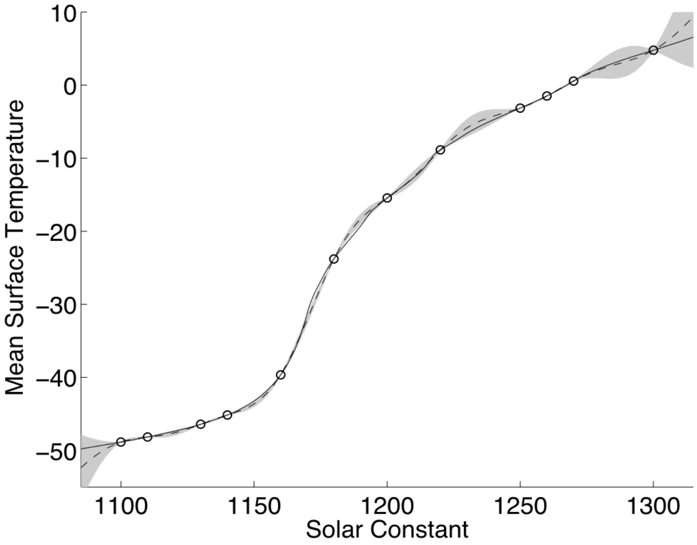


Fig. 12: **Figure 8:** Mean and 95% confidence intervals for the final build of the emulator

Design

For this example, the input ranges we are interested in are $\tilde{X}_1 \in [1370, 1420]$ for the solar constant and $\tilde{X}_2 \in [0.2, 0.4]$ for the albedo. We obtain our initial design from an $p = 2$-dimensional optimised Latin Hypercube in $[0, 1]$, as described in the procedure page for generating an optimised Latin hypercube design (*ProcOptimalLHC*). Figure 1 shows the design points mapped in the input space of the model. The $n = 30$ points in $[0, 1]$ are arranged in a $(p \times n)$ matrix which we call D.

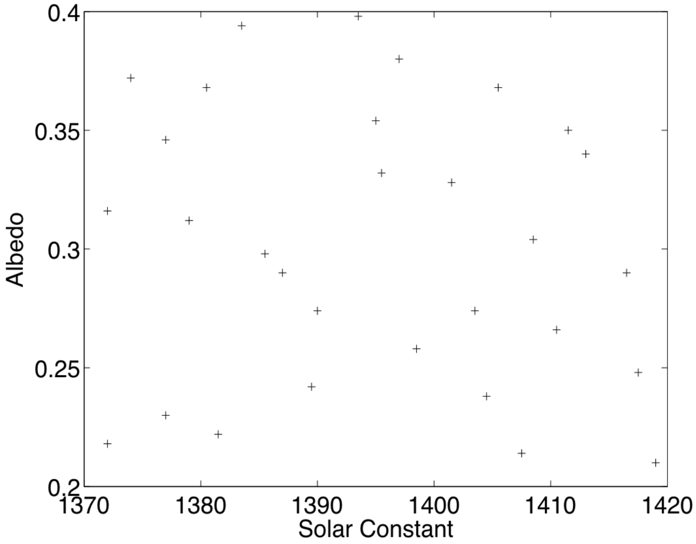


Fig. 13: **Figure 1:** Design points in the simulator's input space

Having obtained our design points we then run the surfebm model at these points and get the mean surface temperature, which we denote as $f(D)$.

Gaussian Process setup

As with the *one dimensional example*, we first set up the Gaussian Process, by selecting a mean and a covariance function. We choose the linear mean function, described in the alternatives page on emulator prior mean function (*AltMeanFunction*), which is $h(x) = [1, x]^{\mathrm{T}}$; this is a $(q \times 1)$ vector, with $q = 1 + p = 3$.

For the covariance function we choose $\sigma^2 c(\cdot, \cdot)$, where the correlation function $c(\cdot, \cdot)$ has the Gaussian form described in the alternatives page on emulator prior correlation function (*AltCorrelationFunction*):

$$c(x, x') = \exp\left[-\sum_{i=1}^{p}\{(x_i - x'_i)/\delta_i\}^2\right]$$

where the subscripts i denote the input.

Estimation of the correlation lengths

The first step in fitting the emulator is to estimate the correlation lengths δ. We do this by first reparameterising δ as $\tau = 2\ln(\delta)$. This makes the optimisation problem unconstrained, because $\tau \in [-\infty, \infty]$, whereas $\delta \in [0, \infty]$. We then find the maximum of the posterior $\pi^*_\tau(\tau)$ assuming a uniform prior on τ, i.e. $\pi_\tau(\tau) \propto \text{const}$. Substitution of $\delta = \exp(\tau/2)$ in $\pi^*_\delta(\delta)$ from the procedure page on building a Gaussian process emulator for the core problem (*ProcBuildCoreGP*) yields

$$\pi^*_\tau(\tau) \propto (\widehat{\sigma}^2)^{-(n-q)/2}|A|^{-1/2}\|H^{\mathrm{T}}A^{-1}H|^{-1/2}.$$

with

$$\widehat{\sigma}^2 = (n - q - 2)^{-1} f(D)^{\mathrm{T}}\left\{A^{-1} - A^{-1}H\left(H^{\mathrm{T}}A^{-1}H\right)^{-1}H^{\mathrm{T}}A^{-1}\right\}f(D).$$

H is the $(n \times p)$ matrix $H = [h(x_1), h(x_2), \cdots, h(x_n)]^{\mathrm{T}}$, with x_n denoting the nth design point.

A is the $n \times n$ correlation matrix of the design points, with elements $[A]_{i,j} \equiv c(x_i, x_j)$, $i, j \in \{1, \cdots, n\}$, which after the reparameterisation can be written as

$$c(x_i, x_j) = \exp\left[-\sum_{k=1}^{p}(x_{k,i} - x_{k,j})^2/\exp(\tau_k)\right].$$

with $x_{k,i}$ denoting the $k$tau‘ is the correlation matrix A.

Figure 2 shows the log of the posterior $\pi^*_\tau(\tau)$ as a function of τ_1, which corresponds to the solar constant, and τ_2, which corresponds to the albedo. The maximum of $\ln(\pi^*_\tau(\tau))$ is found at $\hat{\tau} = [-1.4001, -4.4864]$. The respective values of δ are $\hat{\delta} = [0.4966, 0.1061]$.

We should mention here that $\pi^*_\tau(\tau)$ can have a complicated structure, exhibiting several local maxima. Therefore, the optimisation algorithm has to be initialised using several starting points so as to find the location of the global maximum, or alternatively, a global optimisation method (e.g. simulated annealing) can be used.

We proceed with our analysis, using for δ the estimate

$$\hat{\delta} = [0.4966, 0.1061]$$

Substitution in the expression for $\widehat{\sigma}^2$ yields

$$\widehat{\sigma}^2 = 1.0290$$

Finally, the regression coefficients β are found with the formula

$$\hat{\beta} = \left(H^{\mathrm{T}}A^{-1}H\right)^{-1}H^{\mathrm{T}}A^{-1}f(D).$$

with values $\hat{\beta} = [33.5758, 4.9908, -39.7233]$

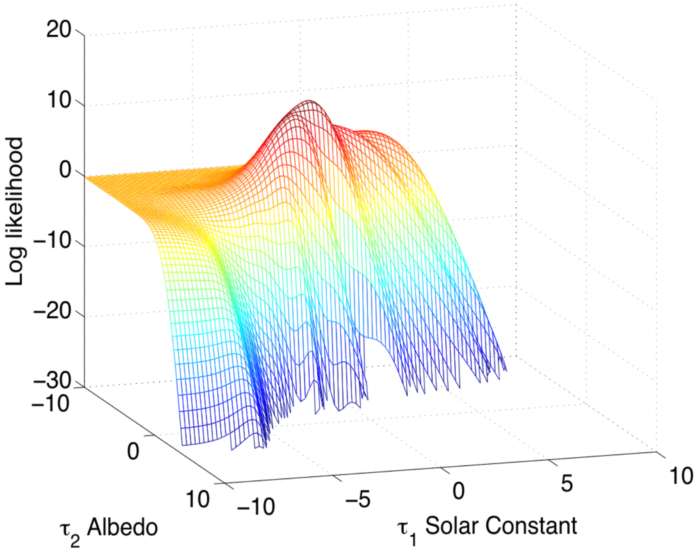


Fig. 14: **Figure 2:** Log posterior $(\ln(\pi^*_\tau(\tau)))$ as a function of τ_1 and τ_2

8.70.3 Validation

After fitting the emulator to the model runs, we need to validate it. We do so by selecting 10 validation points from a Latin hypercube. We call the validation set D'. The validation and the training points are shown in Figure 3.

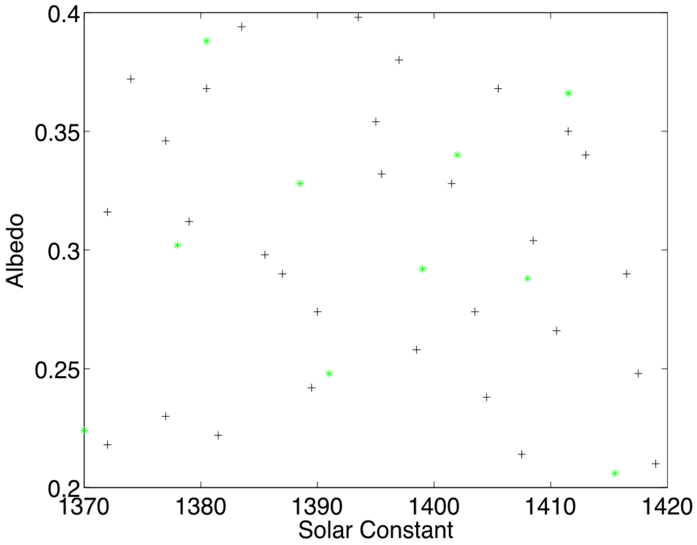


Fig. 15: **Figure 3:** The training (black crosses) and validation points (green stars) in the simulator's input space

We then run the model at the validation points to obtain its output, which we denote by $f(D')$. We finally, estimate the posterior mean $m(D')$ and variance $u(D', D')$, which are given by the formulae in the procedure page for building a Gaussian process emulator for the core problem (*ProcBuildCoreGP*)

$$m^*(x) = h(x)^{\mathrm{T}}\widehat{\beta} + c(x)^{\mathrm{T}}A^{-1}(f(D) - H\widehat{\beta})$$

and

$$v^*(x, x') = \widehat{\sigma}^2\{c(x, x') - c(x)^{\mathrm{T}}A^{-1}c(x') + \left(h(x)^{\mathrm{T}} - c(x)^{\mathrm{T}}A^{-1}H\right)\left(H^{\mathrm{T}}A^{-1}H\right)^{-1}\left(h(x')^{\mathrm{T}} - c(x')^{\mathrm{T}}A^{-1}H\right)^{\mathrm{T}}\}$$

for $x, x' \in D'$.

The individual standardised errors are then estimated, as it is explained in the procedure page for validating a Gaussian process emulator (*ProcValidateCoreGP*). As shown in Figure 4 they all lie within 2 standard deviations.

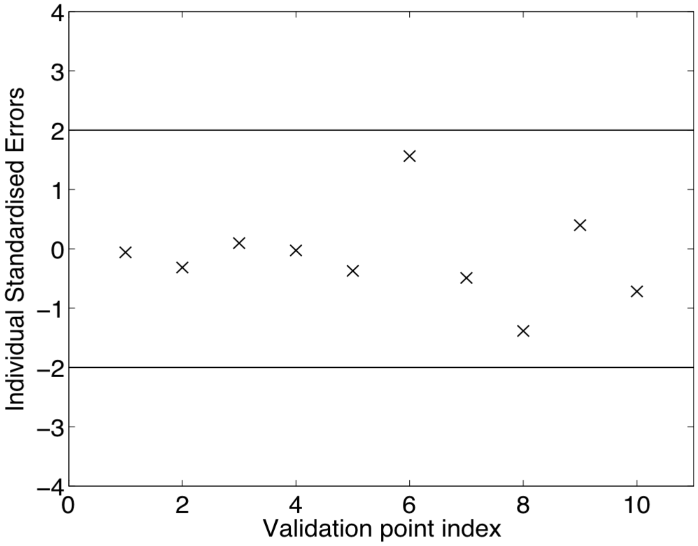


Fig. 16: **Figure 4:** Individual standardised errors

The second validation test we apply is the Mahalanobis distance, also defined in *ProcValidateCoreGP*. Its value is 8.6027, while its theoretical mean is 10 and its standard deviation is 5.5168. Therefore, the estimated Mahalanobis distance, has a value typical of the reference distribution.

Having passed both tests, the emulator is deemed valid. We therefore rebuild the emulator including the validation data, and use this for the subsequent analysis.

Rebuilding the emulator

The design set for the rebuilt emulator contains the previous training points D and the validation points D'. That is,

$$D_{\text{new}} = \{D, D'\}$$

and for simplicity we will denote D_{new} by D in the following. Maximising the posterior $\pi_\tau^*(\tau)$, we find $\hat{\tau} = [-1.2187, -4.6840]$, which corresponds to

$$\hat{\delta} = [0.5437, 0.0961]$$

and by substitution we find

$$\widehat{\sigma}^2 = 0.9354$$
$$\hat{\beta} = [33.5981, 4.8570, -39.6695]^{\mathrm{T}}$$

These parameter values will be used for the uncertainty and sensitivity analysis.

8.70.4 Uncertainty and Sensitivity analysis

Having successfully built the emulator we proceed to perform uncertainty and sensitivity analysis as these are defined in the procedure pages for uncertainty analysis (*ProcUAGP*) and variance based sensitivity analysis (*ProcVarSAGP*) using a GP emulator. We start with the uncertainty analysis.

Uncertainty analysis

The primary goal of uncertainty analysis is to quantify the effect of our uncertainty about the input values of the simulator to its output. We denote by X the model inputs, which are unknown, and we assume that they follow a joint distribution $\omega(X)$. We wish to know what is the effect of this uncertainty about the inputs X on the output $f(X)$. In other words, we seek the mean $\mathrm{E}[f(X)]$ and the variance $\mathrm{Var}[f(X)]$, with respect to the uncertainty distribution $\omega(X)$.

One method of achieving this would be to use a Monte Carlo method. That would involve drawing samples of X from $\omega(X)$ and running the simulator for these input configurations. This method however, requires large computational resources, as we need one simulator run per each Monte Carlo sample. *ProcUAGP* describes how this can be achieved using the emulator. Specifically, *ProcUAGP* describes how to calculate the following measures:

- The posterior mean $\mathrm{E}^*[\mathrm{E}[f(X)]]$
- The posterior variance $\mathrm{Var}^*[\mathrm{E}[f(X)]]$
- The posterior mean $\mathrm{E}^*[\mathrm{Var}[f(X)]]$

In the above, $\mathrm{E}[\cdot]$ and $\mathrm{Var}[\cdot]$ are w.r.t. to $\omega(X)$, and as per usual, $\mathrm{E}^*[\cdot]$ and $\mathrm{Var}^*[\cdot]$ are w.r.t. the emulator.

An interpretation of the above measures could be as follows: the posterior mean $\mathrm{E}^*[\mathrm{E}[f(X)]]$ provides us with an estimate of $\mathrm{E}[f(X)]$, which we could also obtain by running the Monte Carlo based method and averaging the model outputs. The posterior variance $\mathrm{Var}^*[\mathrm{E}[f(X)]]$ is a measure of the uncertainty about the estimate of $\mathrm{E}[f(X)]$. Finally, the posterior mean $\mathrm{E}^*[\mathrm{Var}[f(X)]]$ is an estimate of the variance of $f(X)$, which is due to $\omega(X)$ and could be also estimated by finding the variance of the Monte Carlo model outputs.

The input uncertainty distribution

Before starting, we need to define the uncertainty distribution of the inputs, $\omega(X)$. The definition of this distribution normally requires some interaction with model experts, as discussed in *ProcUAGP*. For the purposes of our example we will use an independent Gaussian distribution for both inputs. More specifically, for both inputs we will use the distribution

$$X_i \sim \mathcal{N}(0.5, 0.02)\,, i \in \{1,2\}$$

which yields for matrix B

$$B = \left[\begin{array}{cc} 50 & 0 \\ 0 & 50 \end{array}\right]$$

and

$$m = \left[\begin{array}{c} 0.5 \\ 0.5 \end{array}\right]$$

Recall that the above distributions refer to the emulator's input space, which is the unit interval $[0,1]$. In the simulator's input space the above distributions are

$$\tilde{X}_1 \sim \mathcal{N}(1395, 50)$$

for the solar constant, and

$$\tilde{X}_2 \sim \mathcal{N}(0.4, 0.0008)$$

for the albedo.

We should also mention that the matrix C is

$$C = \begin{bmatrix} \delta_1^{-2} & 0 \\ 0 & \delta_2^{-2} \end{bmatrix} = \begin{bmatrix} 3.3828 & 0 \\ 0 & 108.2812 \end{bmatrix}$$

Also, since we have not used a nugget, we have $\nu = 0$. Finally, because both B and C are diagonal and our regression term has the linear form $h(x) = [1, x]^{\mathrm{T}}$, we are using the formulae from the special case 2 in *ProcUAGP*.

Results

We now calculate the three measures, $\mathrm{E}^*[\mathrm{E}[f(X)]]$, $\mathrm{Var}^*[\mathrm{E}[f(X)]]$ and $\mathrm{E}^*[\mathrm{Var}[f(X)]]$. In order to do so we first calculate the integral forms $U_p, P_p, S_p, Q_p, U, R, T$ from special case 2, and then use them for the calculation of the above measures according to the formulae given at the top of *ProcUAGP*. The results we got are

$$\mathrm{E}^*[\mathrm{E}[f(X)]] = 16.9857$$
$$\mathrm{Var}^*[\mathrm{E}[f(X)]] = 0.0015$$
$$\mathrm{E}^*[\mathrm{Var}[f(X)]] = 29.9588$$

The above results say that the mean output of our model, due to the uncertainty in its inputs is ~ 16.99, with a $95 : math$:pm 1.96sqrt{0.0015} = pm 0.0387‘. The variance of the output due to the uncertainty from $\omega(X)$ is ~ 29.96.

In order to check our results, we run a Monte Carlo simulation, in which 10000 samples were drawn from $\tilde{X}_1 \sim \mathcal{N}(1395, 50)$ and $\tilde{X}_2 \sim \mathcal{N}(0.4, 0.0008)$, and the model was run for the ensuing input configurations. The result we got was a mean of 17.16 with a 95% confidence interval of 0.11. The variance of the output was estimated as 29.63. Figure 5 shows the distribution of the Monte Carlo output samples, superimposed on the Gaussian $\mathcal{N}(\mathrm{E}^*[\mathrm{E}[f(X)]], \mathrm{E}^*[\mathrm{Var}[f(X)]])$.

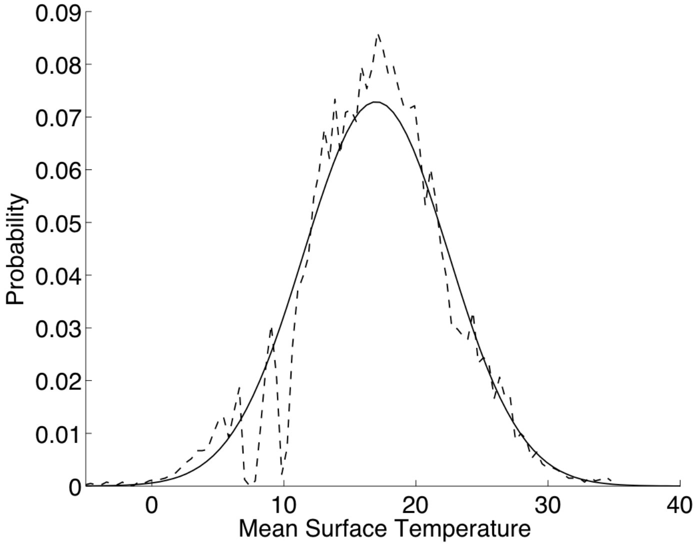


Fig. 17: **Figure 5:** Uncertainty on the output due to $\omega(X)$, as predicted by the emulator (continuous line) and by using Monte Carlo (dashed line)

When considering the above results, we should also keep in mind the improvement in speed that the Bayesian uncertainty methods achieve. In our implementation, the surfebm model took 0.12 seconds for a single run, and as a result the Monte Carlo simulation took approximately 20 minutes. The emulator based method on the other hand, run almost instantaneously, in less than 0.01 seconds.

Sensitivity analysis

In this section we perform *Variance Based Sensitivity Analysis* as described in *ProcVarSAGP*. The main objective of sensitivity analysis is to quantify the effect of our uncertainty about a set of inputs on the output, when the values of the remaining inputs are known. Consequently, it provides a method for assessing the relative importance of the different inputs to the model's output.

The measures we consider here are the average effect $\mathrm{E}^*[M_w(x_w)]$ and the sensitivity index $\mathrm{E}^*[V_w]$, with w being the set of indices of the inputs whose value is known, i.e. $X_w = x_w$. The average effect $\mathrm{E}^*[M_w(x_w)]$ is the difference between the mean of the output when we know the values of the inputs X_w, and the mean of the output when we are uncertain about all the inputs. Therefore, the closer $\mathrm{E}^*[M_w(x_w)]$ is to zero, the smaller the impact the inputs X_w have on the output.

The sensitivity index $\mathrm{E}^*[V_w]$ quantifies the amount by which the variance of the output will decrease if we were to learn the true value of X_w. Therefore, a large sensitivity index suggests that the inputs X_w contribute significantly to the output and vice versa. A somewhat subtle point here is that $\mathrm{E}^*[M_w(x_w)]$ is a function of the value that the inputs X_w take, which we denote by x_w, while $\mathrm{E}^*[V_w]$ is not because the values x_w are averaged out.

Results

In order to estimate the sensitivity measures, we first need to estimate the integral forms described in special case 2 of *ProcVarSAGP*, which are $U_w, P_w, S_w, Q_w, R_w, T_w$ and U, P, S, Q, R, T. Note that U, R, T are the same as those used in the uncertainty analysis section and we do not need to recalculate them. Similarly, $\omega(X), B, m, C$ and ν are also the same.

Having found the values for the integral forms we can then calculate the two measures using the formulae given at the top of *ProcVarSAGP*. Figure 6 shows $\mathrm{E}^*[M_1(x_1)]$ and $\mathrm{E}^*[M_2(x_2)]$. We see that $\mathrm{E}^*[M_1(x_1)]$ is significantly closer to zero than $\mathrm{E}^*[M_2(x_2)]$ is. This implies that changing the value of X_1 (solar constant), does not alter significantly the response of the model. The opposite is not true, because $\mathrm{E}^*[M_2(x_2)]$ varies widely as the value of X_2 changes. We could then conclude that the model's output is significantly more sensitive to the value of the albedo, compared to the value of the solar constant.

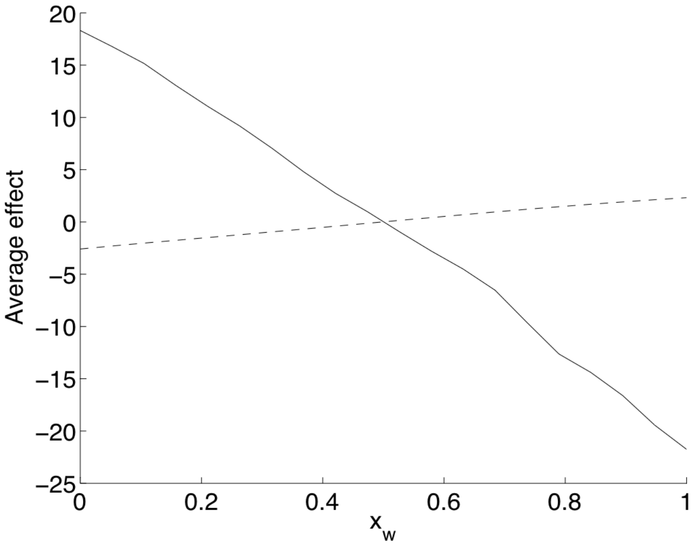


Fig. 18: **Figure 6:** The average effects $\mathrm{E}^*[M_w(x_w)]$ for $w = 1$ (solar constant, dashed line) and $w = 2$ (albedo, continuous line)

The values we calculated for the sensitivity indices are

$$\mathrm{E}^*[V_1] = 0.54$$
$$\mathrm{E}^*[V_2] = 29.40$$

The results show that when we know the value of X_1, the uncertainty (variance) in the output decreases by 0.54, while when we know the value of X_2 it decreases by 29.40. Recall that the variance of the output when we are uncertain about both X_1 and X_2 was found in the previous section to be 29.96. We can then conclude that if we learn the true value of the albedo, our uncertainty about the model's output will decrease by approximately 98%, while learning the true value of the solar constant will decrease our uncertainty only by the remaining 2%.

8.70.5 The model

We finally present in Figure 7 the model that we emulated in this example, along with the design points used for building the emulator, shown with black asterisks. Figure 7 indicates that the model's response depends heavily on the value of the albedo, while the solar constant has a far smaller effect. This intuition is in agreement with the results provided by the sensitivity analysis.

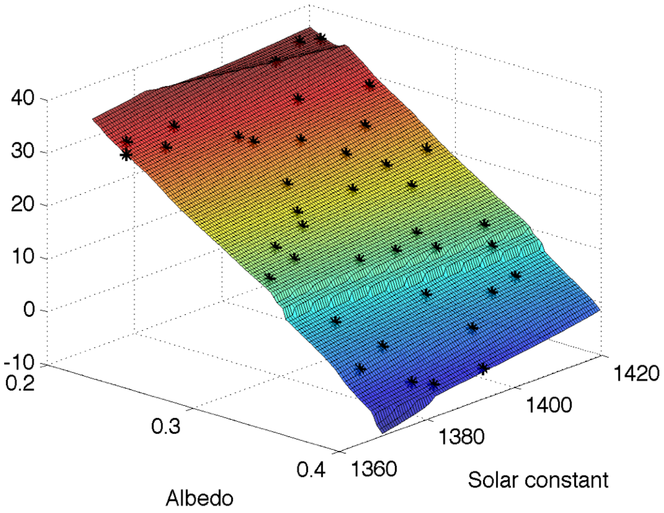


Fig. 19: **Figure 7:** Simulator's output (mean surface temperature) as a function of the solar constant and the albedo. Black asterisks denote the design points D

8.70.6 Data

For those wishing to replicate the above results, we provide the values of the design points and the respective model outputs used in this example. The triplets below correspond to X_1 (solar constant), X_2 (albedo) and $f(X)$ (mean surface temperature).

Original training data

Solar Constant	0.86	0.27	0.57	0.5	0.14	0.23	0.39	0.93	0.98	0.54
Albedo	0.7	0.97	0.29	0.77	0.73	0.11	0.21	0.45	0.05	0.9
Mean Surface Temperature	11.81	-4.95	25.51	5.27	4.85	30.28	27.50	20.89	34.50	0.43
Solar Constant	0.77	0.81	0.18	0.71	0.31	0.04	0.69	0.4	0.63	0.08
Albedo	0.52	0.33	0.56	0.84	0.49	0.09	0.19	0.37	0.64	0.86
Mean Surface Temperature	17.59	25.27	13.39	3.67	16.50	29.96	29.78	21.14	12.78	-0.98
Solar Constant	0.34	0.04	0.83	0.21	0.67	0.47	0.14	0.51	0.75	0.95
Albedo	0.45	0.58	0.75	0.84	0.37	0.99	0.15	0.66	0.07	0.24
Mean Surface Temperature	17.99	12.11	9.10	1.03	22.71	-4.61	28.33	11.60	34.10	29.28

Validation data

Solar Constant	0.00	0.83	0.16	0.37	0.76	0.64	0.58	0.91	0.42	0.21
Albedo	0.12	0.83	0.51	0.64	0.44	0.70	0.46	0.03	0.24	0.94
Mean Surface Temperature	28.66	4.68	15.04	11.63	20.39	10.83	18.78	34.76	26.60	-4.08

Recall that the uncertainty and sensitivity analysis was carried out using both data sets above as training data.

8.71 Example: Illustrations of decision-based sensitivity analysis

8.71.1 Description and background

Decision-based sensitivity analysis (SA) is a form of probabilistic SA; see the page on varieties of SA (*DiscWhyProbabilisticSA*). We present here some examples of the formulation of decision-based sensitivity analyses. Relevant notation and definitions are given in the decision-based SA page (*DiscDecisionBasedSA*), but we repeat some key definitions here for convenience.

In probabilistic SA, we regard the *simulator* inputs X as uncertain with probability distribution $\omega(x)$. In decision-based SA, we further require a decision set $\mathcal{D}$ whose elements are the possible decisions, and a loss function $L(d,x)$ expressing the consequences of taking decision $d \in \mathcal{D}$ when the true inputs are $X = x$. The optimal decision is the one having smallest expected loss

$$\bar{L}(d) = \mathrm{E}[L(d,X)],$$

where the expectation is taken with respect to the distribution of X. Therefore if the optimal decision is M,

$$\bar{L}(M) = \min_d\{\bar{L}(d)\}.$$

If we were able to learn the true value of a subset x_J of the inputs, then the optimal decision would minimise

$$\bar{L}_J(d,x_J) = \mathrm{E}[L(d,X)\,|\,X_J = x_J].$$

and would depend on x_J. If we denote this decision by $M_J(x_J)$, then the decision-based sensitivity measure for X_J is the expected value of information

$$V_J = \bar{L}(M) - \mathrm{E}[\bar{L}_J(M_J(X_J),X_J)].$$

Our examples illustrate some decision problems that can be formulated through $\mathcal{D}$ and $L(d,x)$, and explore the nature of V_J in each case.

8.71.2 Example 1: Estimation with quadratic loss

First suppose that the decision problem is to estimate the true simulator output $f(x)$. Hence the decision set $\mathcal{D}$ consists of all possible values of the simulator output $f(x)$. Our loss function should be of the form that there is no loss if the estimate (decision) exactly equals the true $f(X)$, and should be an increasing function of the estimation error $f(x)-d$. One such function is the quadratic loss function

$$L(d,x) = \{f(x) - d\}^2.$$

If we now apply the above theory, the optimal estimate d will minimise

$$\begin{aligned}\bar{L}(d) &= \mathrm{E}[f(X)^2 - 2d\,f(X) + d^2] = \mathrm{E}[f(X)^2] - 2d\,\mathrm{E}[f(X)] + d^2 \\ &= \mathrm{Var}[f(X)] + \{\mathrm{E}[f(X)] - d\}^2\,,\end{aligned}$$

after simple manipulation using the fact that, for any random variable Z, $\mathrm{Var}[Z] = \mathrm{E}[Z^2] - \{\mathrm{E}[Z]\}^2$. We can now see that this is minimised by setting $d = \mathrm{E}[f(X)]$. We can also see that the expected loss of this optimal decision is $\bar{L}[M] = \mathrm{Var}[f(X)]$. In this decision problem, then, we find that M has the same form as defined for *variance-based SA* in the page *DiscVarianceBasedSA*. By an identical argument we find that if we learn the value of x_J the optimal decision becomes $M_J(x_J) = \mathrm{E}[f(X)\,|\,x_J]$, which is also as defined in *DiscVarianceBasedSA*. Finally, the sensitivity measure becomes

$$V_J = \mathrm{Var}[f(X)] - \mathrm{E}[\mathrm{Var}[f(X)\,|\,X_J]],$$

which is once again the sensitivity measure defined in *DiscVarianceBasedSA*.

This example demonstrates that we can view variance-based SA as a special case of decision-based SA, in which the decision is to estimate $f(x)$ and the loss function is quadratic loss (or squared error).

Before finishing this example it should be noted first that *DiscVarianceBasedSA* presents a rationale for variance-based SA that is persuasive when there is no underlying decision problem. But second, if there genuinely is a decision to estimate $f(x)$ the loss function should be chosen appropriately to the context of that decision. It might be quadratic loss, but depending on what units we choose to measure that loss in we might have a multiplier c such that $L(d,x) = c\{f(x)-d\}^2$. The optimal decisions would be unchanged but the sensitivity measure (the value of information) would now be multiplied by c.

The same applies to any decision problem. Making a linear transformation of the loss function does not change the optimal decision, but the value of information is multiplied by the scale factor of the transformation.

8.71.3 Example 2: Optimisation

A common decision problem in the context of using a simulator is identify the values of certain inputs in order to maximise or minimise the output. We will suppose here that the objective is to minimise $f(x)$, so that the output itself serves as the loss function. This is straightforward if all of the inputs are under our control, but the problem becomes more interesting when only some of the inputs can be controlled to optimise the output, while the remainder are uncertain. We therefore write the simulator as $f(d,y)$, where d denotes the control inputs and y the remaining inputs. The latter are uncertain with distribution $\omega(y)$.

So the decision problem is characterised by a decision set $\mathcal{D}$ comprising all possible values of the control inputs d, together with a loss function

$$L(d,x) = f(x) = f(d,y).$$

To illustrate the calculations in this case, let the simulator have the form

$$f(d,y) = y_1\{y_2 + (y_1 - d)^2\},$$

where d is a scalar and $y = (y_1, y_2)$. With y uncertain, we compute the expected loss. Simple algebra gives

$$\bar{L}(d) = \mathrm{E}[Y_1 Y_2 + Y_1^3 - 2dY_1^2 + d^2 Y_1] = \mathrm{E}[Y_1](d - M)^2 + \bar{L}[M],$$

where

$$M = \mathrm{E}[Y_1^2]/\mathrm{E}[Y_1]$$

is the optimal decision and

$$\bar{L}[M] = \mathrm{E}[Y_1 Y_2] + \mathrm{E}[Y_1^3] - \{\mathrm{E}[Y_1^2]\}^2/\mathrm{E}[Y_1]$$

is the minimal expected loss, i.e. the expected value of the simulator output at $d = M$. If now we were to learn the value of y_1, then

$$\bar{L}_1(y_1) = \mathrm{E}[L(d, X) \,|\, y_1] = y_1\{\mathrm{E}[Y_2 \,|\, y_1] + (y_1 - d)^2\},$$

and the optimal decision would be $d = y_1$ with minimal (expected) loss $y_1 \mathrm{E}[Y_2|y_1]$. Since the expectation of this with respect to the uncertainty in y_1 is just $\mathrm{E}[Y_1 Y_2]$, we then find that the sensitivity measure for y_1, i.e. the expected value of learning the true value of y_1, is

$$V_{\{1\}} = \mathrm{E}[Y_1^3] - \{\mathrm{E}[Y_1^2]\}^2/\mathrm{E}[Y_1].$$

It is straightforward to see that the value of learning both y_1 and y_2 is the same, $V_{\{1,2\}} = V_{\{1\}}$, because the optimal decision is still $d = y_1$. So if we could learn y_1 there would be no additional value in learning y_2.

It remains to consider the sensitivity measure for learning y_2. We now find that the optimal decision is

$$M_2(y_2) = \mathrm{E}[Y_1^2 \,|\, y_2]/\mathrm{E}[Y_1 \,|\, y_2]$$

and the expected value of learning the true value of y_2 is

$$V_{\{2\}} = \mathrm{E}\left[\{\mathrm{E}[Y_1^2 \,|\, Y_2]\}^2/\mathrm{E}[Y_1 \,|\, Y_2]\right] - \{\mathrm{E}[Y_1^2]\}^2/\mathrm{E}[Y_1].$$

If the two uncertain inputs are independent, then we find that this reduces to zero. Otherwise, learning the value of y_2 gives us some information about y_1, which in turn has value.

Numerical illustration. Suppose that Y_2 takes the value 0 or 1 with equal probabilities and that the distribution of Y_1 given Y_2 is $\mathrm{Ga}(2, 5 + y_2)$, with moments $\mathrm{E}[Y_1|y_2] = (5 + y_2)/2$, $\mathrm{E}[Y_1^2|y_2] = (5 + y_2)(6 + y_2)/4$ and $\mathrm{E}[Y_1^3|y_2] = (5 + y_2)(6 + y_2)(7 + y_2)/8$. Then

$$\mathrm{E}[Y_1] = 0.5(5/2 + 6/2) = 2.75.$$

We similarly find that $\mathrm{E}[Y_1^2] = 9$ and $\mathrm{E}[Y_1^3] = 34.125$ and hence that $V_{\{1,2\}} = V_{\{1\}} = 4.67$. In contrast we find $V_{\{2\}} = 0.17$, a small value that reflects the limited way that learning about y_2 provides information about y_1.

Before ending this example we consider an additional complication. As explained in the page describing uses of SA in the toolkit (*DiscToolkitSensitivityAnalysis*), in the context of complex, computer-intensive simulators we will have additional *code uncertainty* arising from building an *emulator* of the simulator. All of the quantities required in the decision-based SA calculation are now subject to code uncertainty. Two approaches to incorporating code uncertainty are discussed in *DiscToolkitSensitivityAnalysis*.

In the approach characterised as decision under code uncertainty, we optimise the posterior expected loss $\bar{L}^*(d) = \mathrm{E}^*[\bar{L}(d)]$, where $\mathrm{E}^*[\cdot]$ denotes a posterior expectation (in the case of a fully *Bayesian* emulator, or the adjusted mean in the *Bayes linear* case). In this example, $\bar{L}^*(d)$ is the expectation of the posterior mean of $f(d, Y)$. We simply replace the emulator by its posterior mean and carry out all the computations above.

However, it is often more appropriate to consider the second approach that is characterised as code uncertainty about the decision. Code uncertainty now induces a (posterior) distribution for M whose mean is not generally just the result of minimising $\bar{L}^*(d)$. In general, the analysis for the optimisation problem becomes much more complex in this approach.

8.72 Example: A one-input, two-output emulator

8.72.1 Introduction

In this page we present an example of fitting multivariate emulators to a simple 'toy' simulator that evaluates a pair of polynomial functions on a one dimensional input space. We fit emulators with four different types of covariance function:

- an independent outputs covariance function;
- a separable covariance function;
- a convolution covariance function;
- a LMC covariance function.

These covariance functions are described in full in the alternatives page *AltMultivariateCovarianceStructures*.

8.72.2 Simulator description

The simulator we use has $p = 1$ input and $r = 2$ outputs. It is is a 'toy' simulator that evaluates a pair of ninth degree polynomial functions over the input space $[0, 1]$. Figures 1(a) and 1(b) show the outputs of the simulator plotted against the input.

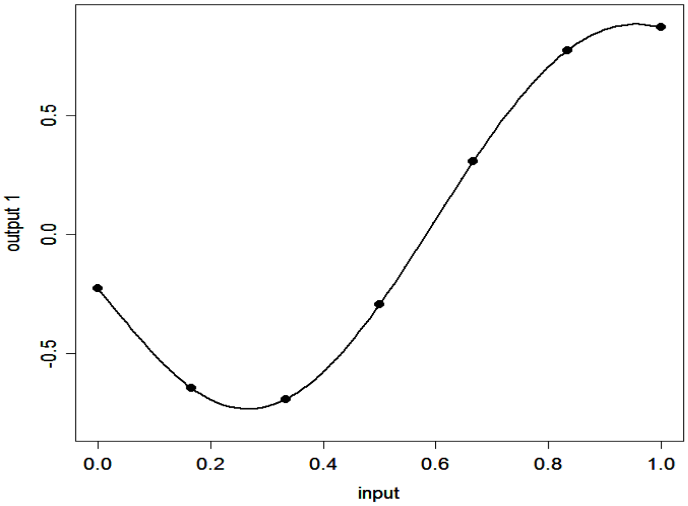


Fig. 20: **Figure 1(a):** Output 1 of the simulator and the 7 design points

8.72.3 Design

Design is the selection of the input points at which the simulator is to be run. There are several design options that can be used, as described in the alternatives page on training sample design for the core problem (*AltCoreDesign*). For this example, we will simply use a set of $n = 7$ equidistant points on the space of the input variable,

$$D = (0, 0.1666667, 0.3333333, 0.5, 0.6666667, 0.8333333, 1) .$$

The simulator output at these points is

$$f_1(D) = (-0.2240000, -0.6442619, -0.6907758, -0.2931373, 0.308774, 0.774162, 0.870681)\,,$$
$$f_2(D) = (-0.2208932, -0.4321548, -0.6921707, 0.02779835, 0.2002072, 0.7119241, 0.7743888)\,,$$

The true functions computed by the simulator, and the design points, are shown in Figure 1.

8.72.4 Multivariate Gaussian Process setup

In setting up the Gaussian process, we need to define the mean and the covariance function. As mentioned in the introduction we fit four different emulators, each with a different covariance function chosen from the options listed in *AltMultivariateCovarianceStructures*. They are:

- an independent outputs covariance function with Gaussian form correlation functions;
- a separable covariance function with a Gaussian form correlation function;
- a convolution covariance function with Gaussian form smoothing kernels;
- a LMC covariance function with with Gaussian form basis correlation functions.

We refer to the emulators that result from these four choices as IND, SEP, CONV and LMC respectively.

In all four emulators we assume we have no prior knowledge about how the output will respond to variation in the inputs, so choose a constant mean function as described in *AltMeanFunction*, which is $h(x) = 1$ and $q = 1$.

8.72.5 Estimation of the hyperparameters

Each emulator has one or more hyperparameters related to its covariance function, which require a prior specification. The hyperparameters and the priors we use are as follows:

- IND
  - Hyperparameters: $\delta = (\delta_1, \delta_2)$, 2 correlation lengths.
  - Prior: $\pi_\delta(\delta) = \pi_{\delta_1}(\delta_1)\pi_{\delta_2}(\delta_2) \propto \delta_1^{-3}\delta_2^{-3}$. This corresponds to independent noninformative (flat) priors on the inverse squares of the elements of δ.
- SEP
  - Hyperparameter: δ, the correlation length.
  - Prior: $\pi_\delta(\delta) \propto \delta^{-3}$. This corresponds to a noninformative (flat) prior on the inverse square of δ.
- CONV
  - Hyperparameters: $\omega = (\delta, \tilde{\Sigma})$, where $\delta = (\delta_1, \delta_2)$ are 2 correlation lengths for the smoothing kernels.
  - Prior: $\pi_\omega(\omega) \propto \delta_2^{-3}\delta_1^{-3}|\tilde{\Sigma}|^{-3/2}$. This corresponds to independent noninformative priors on $\tilde{\Sigma}$ and the inverse squares of the elements of δ.
- LMC
  - Hyperparameter: $\omega = (\tilde{\delta}, \Sigma)$, where $\tilde{\delta} = (\tilde{\delta}_1, \tilde{\delta}_2)$ are 2 basis correlation lengths, and Σ, the between outputs covariance function.
  - Prior: $\pi_\omega(\omega) \propto \tilde{\delta}_2^{-3}\tilde{\delta}_1^{-3}|\Sigma|^{-3/2}$. This corresponds to independent noninformative priors on Σ and the inverse squares of the elements of $\tilde{\delta}$.

We estimate the hyperparameters by maximising the posterior. For IND we take the data from each output $i = 1, 2$ in turn and maximise :math:pi^*_{delta_i}(.), the single output GP hyperparameter posterior, as given in *ProcBuildCoreGP<ProcBuildCoreGP>*. For SEP we maximise :math:pi^*_delta(.) as given in *ProcBuildMultiOutputGPSep<ProcBuildMultiOutputGPSep>*. For CONV and LMC we maximise :math:pi_omega(.) as given in *ProcBuildMultiOutputGP<ProcBuildMultiOutputGP>*. The estimates we obtain are as follows:

- IND
  - $(\hat{\delta}_1, \hat{\delta}_2) = (0.6066194, 0.2156990)$
- SEP
  - $\hat{\delta} = 0.1414267$
- CONV
  - $(\hat{\delta}_1, \hat{\delta}_2) = (0.4472136, 0.1777016)$
  - $\hat{\tilde{\Sigma}} = \begin{pmatrix} 0.4091515 & 0.2576867 \\ 0.2576867 & 0.3039197 \end{pmatrix}$
- LMC
  - $(\hat{\tilde{\delta}}_1, \hat{\tilde{\delta}}_2) = (0.4472136, 0.2072804)$
  - $\hat{\Sigma} = \begin{pmatrix} 0.322907165 & 0.006548224 \\ 0.006548224 & 0.254741777 \end{pmatrix}$

8.72.6 Posterior mean and Covariance functions

The expressions for the posterior mean and covariance functions are given in *ProcBuildCoreGP* for IND , in *ProcBuildMultiOutputGPSep* for SEP, and in *ProcBuildMultiOutputGP* for CONV and LMC.

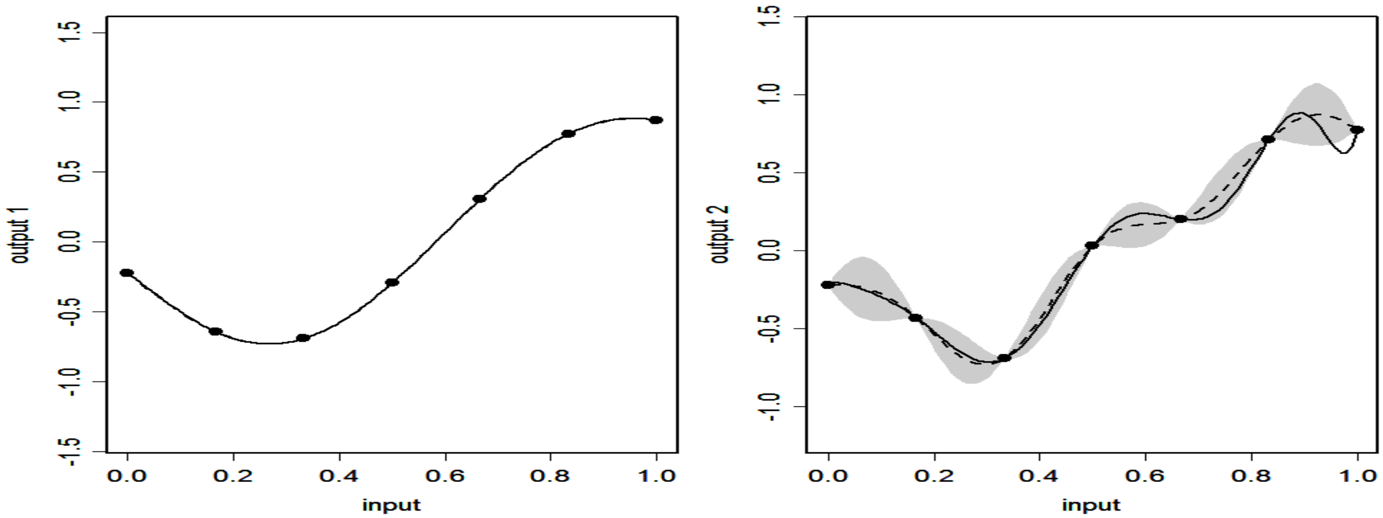


Fig. 21: **Figure 2(a):** IND emulator: Simulator (continuous line), emulator's mean (dashed line) and 95% posterior intervals (shaded area)

Figures 2(a)-2(d) show the predictions of the outputs given by the emulators for 100 points uniformly spaced in $[0, 1]$. The continuous line is the output of the simulator and the dashed line is the emulator's posterior mean $m^*(.)$. The shaded areas represent 2 times the standard deviation of the emulator's prediction, which is the square root of the diagonal of matrix $v^*(., .)$.

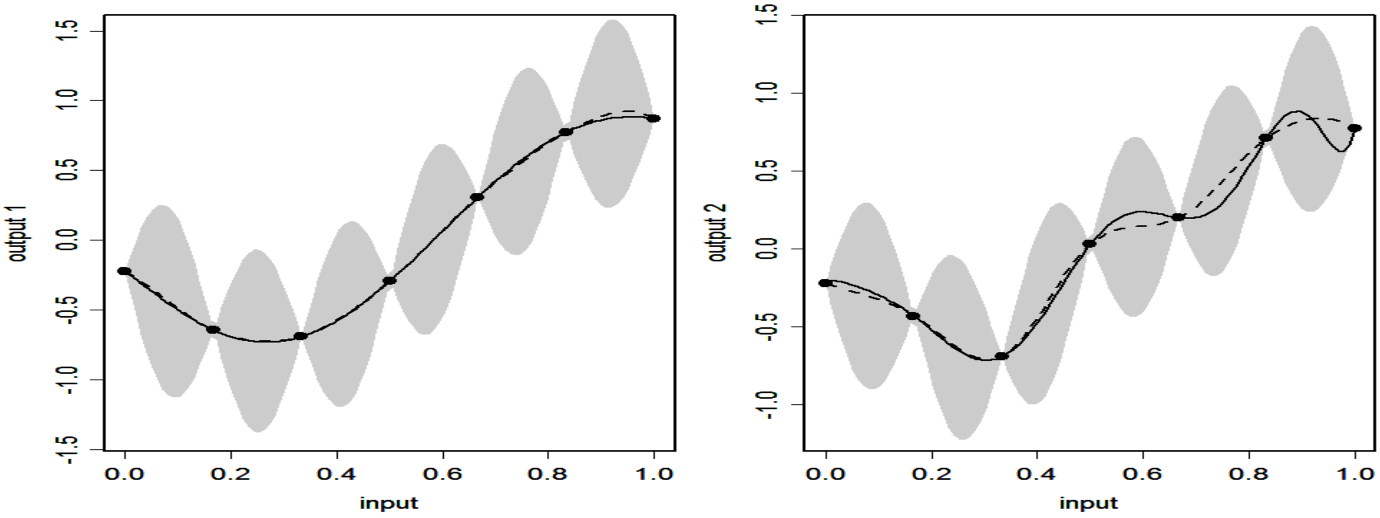


Fig. 22: **Figure 2(b):** SEP emulator: Simulator (continuous line), emulator's mean (dashed line) and 95% posterior intervals (shaded area)

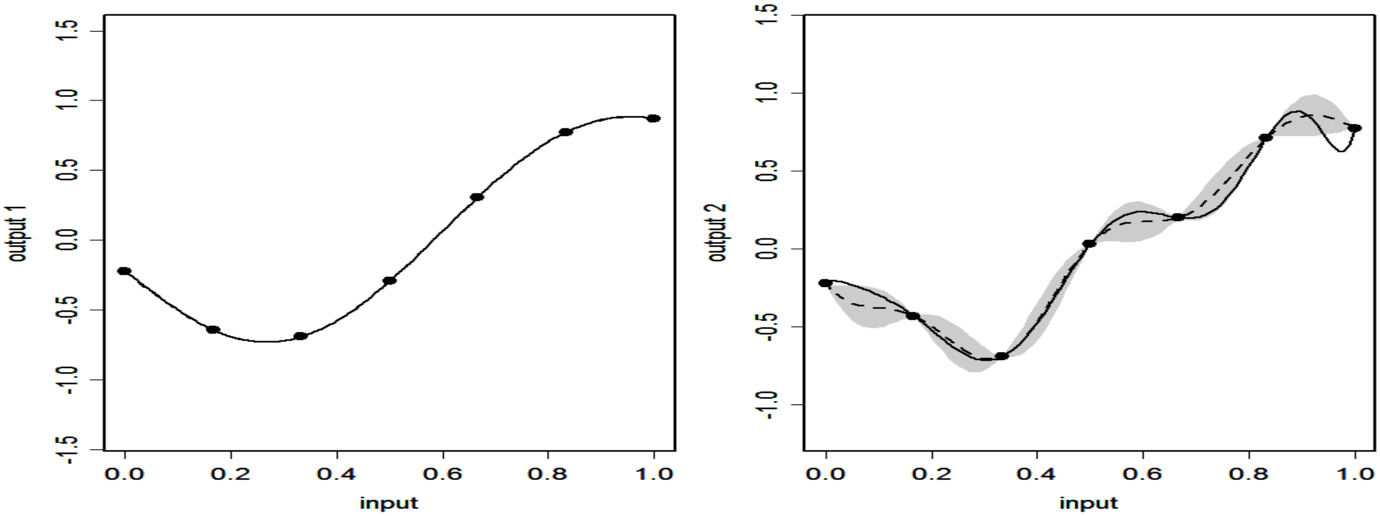


Fig. 23: **Figure 2(c):** CONV emulator: Simulator (continuous line), emulator's mean (dashed line) and 95% posterior intervals (shaded area)

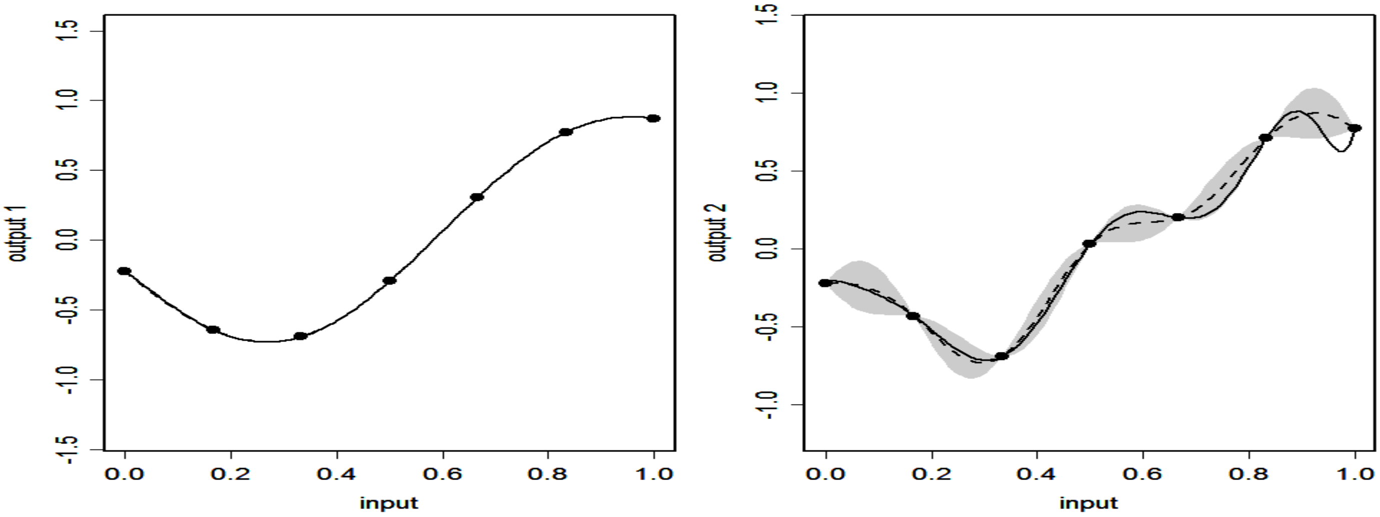


Fig. 24: **Figure 2(d):** LMC emulator: Simulator (continuous line), emulator's mean (dashed line) and 95% posterior intervals (shaded area)

We see that, for all four emulators, the posterior mean matches the simulator almost exactly for output 1, but the match is less good for output 2. This is because output 2 has many turning points, and the data miss several extrema, making prediction of these extrema difficult. The main difference between the emulators is in the widths of the posterior 95% interval. For IND, CONV and LMC the interval for output 1 has almost zero width, which is appropriate since there is very little posterior uncertainty about this output, and the interval for output 2 is wide enough to capture the true function in most regions. SEP, on the other hand, has wide intervals for both outputs. This is because both outputs have the same input space correlation function. While the wide interval is appropriate for output 2, it is not appropriate for output 1 as it suggests much more uncertainty about the predictions than necessary.

Figure 3(a) shows plots of the simulator output and the emulator prediction in the (output 1, output 2)-space at one particular input point, $x = 0.417$. Also shown is an ellipse that represents the 95% highest probability density posterior region. We see that the 95% ellipses are much smaller for IND, CONV and LMC than for SEP. Figure 3(b) shows the same plots, but with different axis scales, in which we see that the 95% ellipse for IND is symmetric in the coordinate directions, while those for SEP, CONV and LMC are rotated. This shows that the non-independent multivariate emulators have non-zero between-output correlation in their posterior distributions.

8.72.7 Discussion

This example demonstrates some of the features of multivariate emulators with a number of different covariance functions. In summary,

- The independent outputs approach can produce good individual output predictions, and can cope with outputs with different smoothness properties. However, it does not capture the between-outputs correlation, which in this example resulted in a poor representation of joint-output uncertainty.
- The multivariate emulator with a separable covariance function may produce poor results when outputs have different smoothness properties. In this example the problem was mostly with the predictive variance. The feature of having just one input space correlation function for both outputs meant that the posterior uncertainty for at least one output was inappropriate.
- The multivariate emulators with nonseparable covariance functions (the convolution covariance and the LMC covariance) can produce good individual output predictions when outputs have different smoothness properties, and can correctly represent joint-output uncertainty.

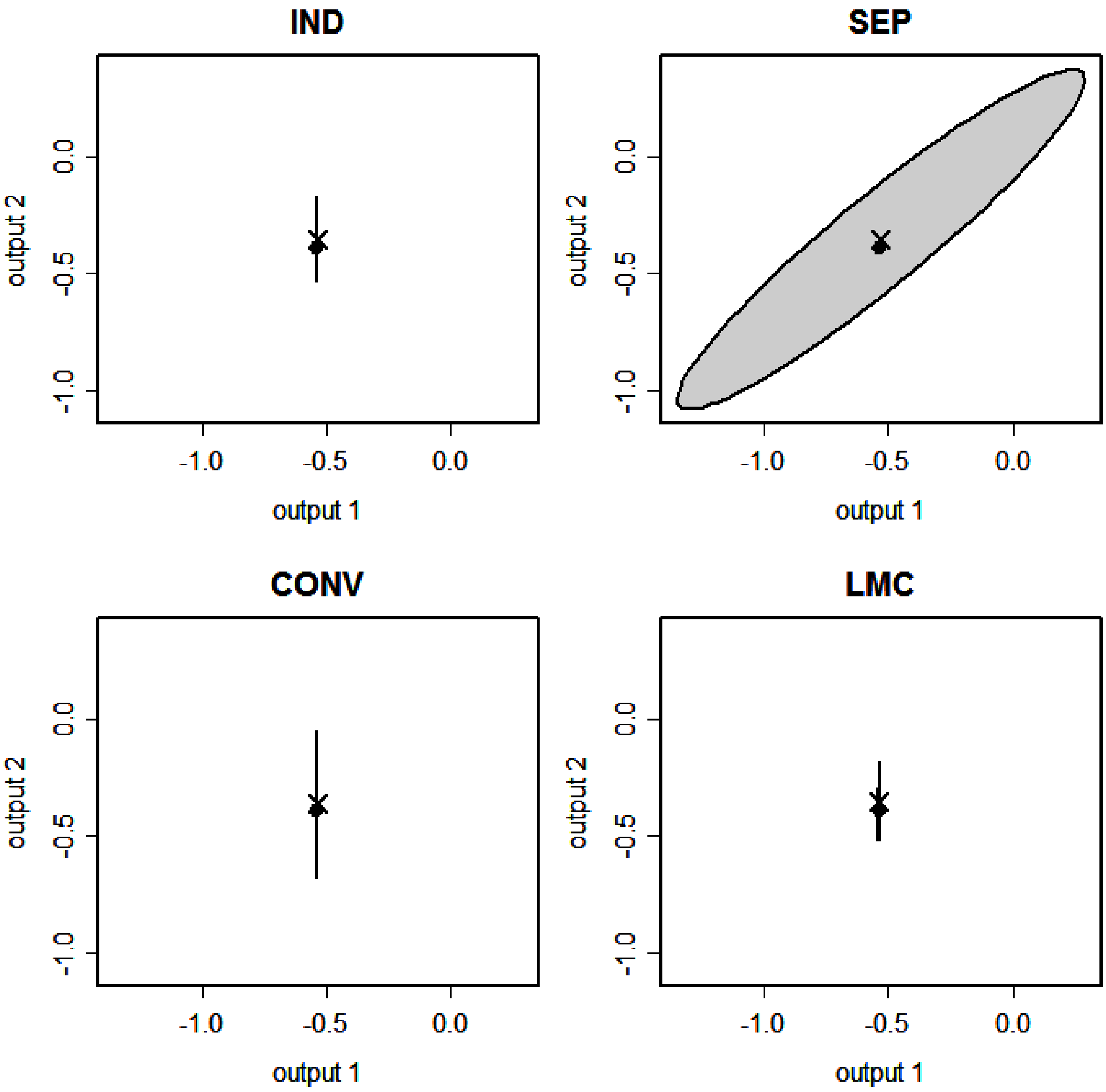


Fig. 25: **Figure 3(a):** Plots of the bivariate output space, showing the simulator output for $x = 0.75$ (black dot), emulator prediction (cross) and 95% posterior region.

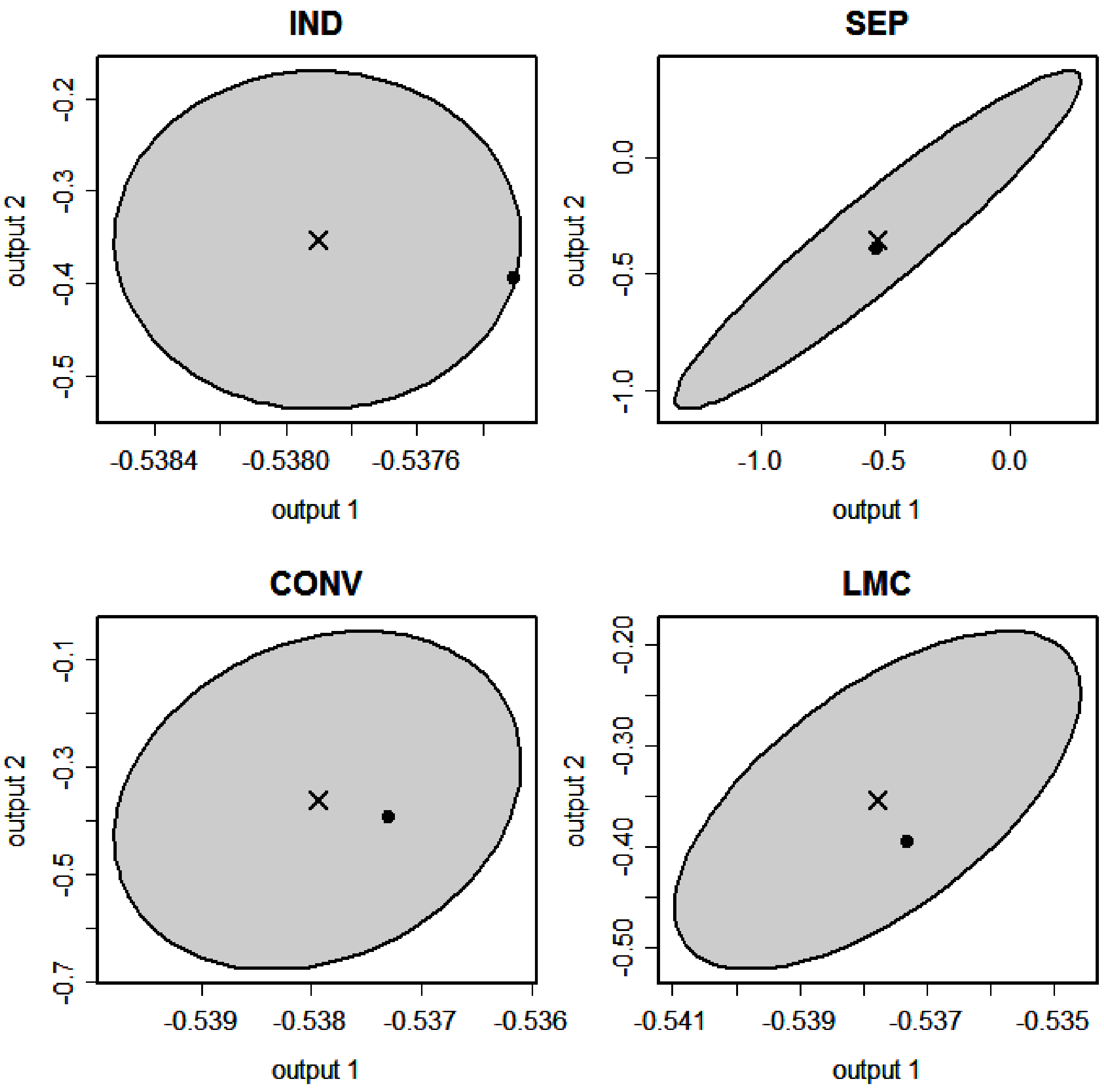


Fig. 26: **Figure 3(b):** The same as Figure 3(a), but with each plot shown on its own axis scales.

This example may suggest that multivariate emulators with nonseparable covariance functions may be the best option in general multi-output problems, since they have the greatest flexibility. However, we must remember this is only a very small-scale example. In larger, more realistic examples, the complexity of nonseparable covariance functions may make them infeasible, due to the large number of hyperparameters that must be estimated. In that case it may come down to a decision between the independent outputs approach (good for individual output predictions) or a multivariate emulator with a separable covariance function (good for representation of joint-output uncertainty).

8.73 Example: Multi-output emulator with separable covariance and dimension reduction

The aim of this example is to illustrate the construction of a multi-output emulator for the chemical reaction simulator described below. The example uses a real model for the decomposition of azoisopropane but does make some simplifying assumptions to enable clearer presentation of the emulation.

8.73.1 Outline

8.73.2 Model (simulator) description

This page provides an example of fitting an *emulator* to a *multi-output simulator*. It follows the structure taken in *the example for a single output case*. The simulator of interest, denoted f, models a chemical reaction in which two initial chemicals react creating new chemical species in the process, while themselves being partially consumed. These new species in turn interact with each other and with the initial reactants to give further species. At the end of the reaction, identified by a stable concentration in all reactants, 39 chemical species can be identified.

The system is deterministic and assumed to be controlled by $p = 12$ inputs: the temperature, the initial concentrations of the 2 reactants and 9 reaction rates controlling the 9 principal reactions. Other reaction rates, for reactions which are present in the system are assumed to be known precisely and are thus fixed - the main reason for doing this is to constrain the number of parameters that need to be estimated in building the emulator. In theory, if this were being done in a real application it would be beneficial to undertake some initial *screening* to identify the key reactions, if these were not known a priori. The outputs of the simulator are the $r = 39$ final chemical concentrations, measured in log space, which is a common transformation for concentrations which must be constrained to be positive.

8.73.3 Design

We use a *space-filling design* (maxi-min latin hypercube) over suitable ranges of temperature, initial concentrations of the two species and the 9 key reaction rates. Given the input and output dimensions of the simulator, and therefore emulator, we need a sufficiently large *training sample*. This is generated by taking a sample of $n = (r+p)*10 = 510$ input points within valid ranges. This choice is somewhat arbitrary, but smaller training set sizes were empirically found to produce less reliable emulators. The valid range for each input is taken to be +/-20% of the most likely value specified by the expert. The sample inputs are collected in the design matrix D and the simulator is evaluated at each point in D, yielding the training outputs $f(D)$.

8.73.4 Preprocessing

Inputs

The inputs are normalised, so that all input points lie in the unit cube $[0,1]^p$. This is to avoid numerical issues to do with the respective scale of the different inputs (temperatures vary over order 100K while concentrations vary over order 1e-4).

Outputs

Following the procedure page for a principal component transformation of the outputs (*ProcOutputsPrincipalComponents*), we apply Principal Component Analysis to reduce correlations between outputs, and reduce the dimension of the output domain. However here the aim of using principal components analysis is somewhat different from the reason described in *ProcOutputsPrincipalComponents*. In *ProcOutputsPrincipalComponents* the aim is the produce a set of outputs that are as uncorrelated as possible so that each output can be modelled by a separate emulator, as in the generic thread for combining two or more emulators (*ThreadGenericMultipleEmulators*). In this example the aim is to reduce the output dimensions that need to be emulated, to simplify the emulation task. It may also be beneficial to render the outputs less correlated, but this is not critical here, since a *truly multivariate Gaussian process emulator* is used.

We first subtract the mean from the data $F = f(D) - E[f(D)]$ and compute the covariance matrix of the residuals F. We then compute the eigenvalues and eigenvectors of $\mathrm{cov}(F)$.

Figure 1 shows the eigen-spectrum of the outputs estimated from the 510 training simulator runs.

The first 10 components (10 largest eigenvalues) explain 99.99% of the variance.

However care must be taken here since the outputs are all very small concentrations expressed in log-space. This means that the lower concentrations are likely to have a larger variance (because of their large negative log-values) than the higher concentrations. Hence, emulating well even the smaller principal components is important. There are several other approaches that could have been taken here, for example it would have been possible to normalise all outputs to lie in $[0,1]^r$, which might make sense if each output was as important as any other. Other approaches to *output transformations* could be considered, for example changing the transformation, to for example a square root transformation, or a Box-Cox transformation to provide something closer to what is felt to be the important aspect of the simulator to be captured.

In this example we choose to retain $r^- = 15$ components, rather than the 10 that are apparently sufficient. We then project the training outputs $f(D)$ onto these first r^- components, yielding the reduced training outputs $f^-(D) = f(D) timesP$, where P is the $r \times r^-$ projection matrix made of the first r^- eigen-vectors.

Note: Projection onto the first 7, 10 and 12 components can be shown to result in poor emulation of the larger concentrations (by retaining 12 principal components, only the largest concentration is incorrectly emulated). All outputs are well emulated with 15 components, as shown below. This is an important point in real applications - think carefully about the outputs you most want to emulate - these might not always appear in the largest eigen-value components.

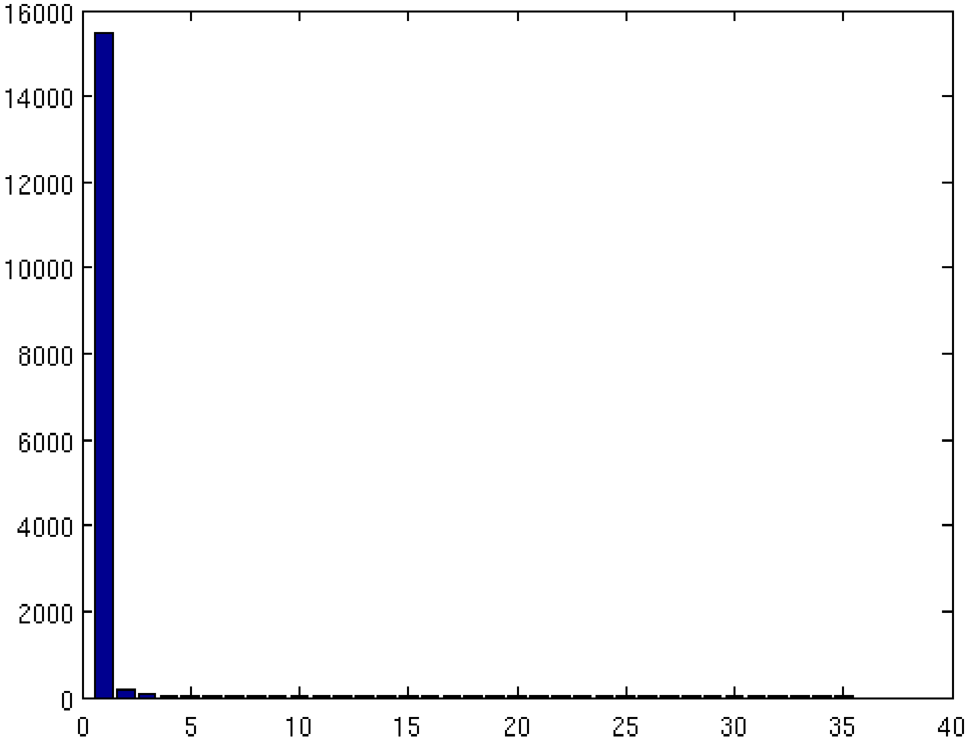


Fig. 27: **Figure 1:** Eigen-values of the principal components on the simulator outputs (based on 510 simulator runs)

Complete emulation process

The complete emulation process (including pre-processing) thus comprises of the following steps:

1. Choose a design for the training inputs, D, and evaluate the simulator, yielding the training outputs $f(D)$
2. Normalise the inputs x, yielding x^-
3. Project the training outputs onto the first r^- principal components, yielding $f^-(D)$
4. Emulate $f^- : x^- \to f^-(D)$
5. Project back onto the r dimensions of the original output

Any further analysis is performed in projected output space unless specified otherwise. We omit the $^-$ in the following to keep the notation concise, however we will now work almost exclusively in the projected outputs space.

8.73.5 Gaussian Process emulator setup

Mean function

In setting up the Gaussian process emulator, we need to define the mean and the covariance function. For the mean function, we choose the linear form described in the alternatives page on the emulator prior mean function (*AltMeanFunction*), with functions $h(x) = [1, x]^T$ and $q = 1 + p = 13$ (x is 12-dimensional) and regression coefficients β. Note the outputs are high dimensional and we assume that each one has its own linear form, meaning there are a lot of regression coefficients β.

Covariance function

To emulate the simulator, we use a *separable multi-output Gaussian process* with a covariance function of the form $\Sigma \otimes c(.,.)$. The covariance between outputs is assumed to be a general positive semi-definite matrix Σ with elements to be determined below. There is no particular reason to assume a separable form is suitable here, and a later example will show the difference between independent , separable and full covariance emulators. The various options for multivariate emulation are discussed in *AltMultipleOutputsApproach*.

The correlation function (between inputs) has a *Gaussian form*:

$$c(x, x') = \exp\left[-\frac{1}{2}(x - x')^T C(x - x')\right] + \nu I$$

with C the diagonal matrix with elements $C_{ii} = 1/\delta_i^2$ with i representing each of the r the (reduced dimension) outputs.

Since the projected training outputs are not a perfect mapping of the original outputs due to the dropped components, we do not want to force the emulator to interpolate these exactly. Thus we add a nugget ν to the covariance function, which accounts for the additional uncertainty introduced by the projection as discussed in the alternatives page on emulator prior correlation function (*AltCorrelationFunction*), and allows the emulator to interpolate the projected training outputs without fitting them exactly. As a side note, adding a nugget also helps to mitigate the computational problems that arise from the use of the Gaussian form in the covariance function. This means that the emulator is no longer an exact interpolation for the simulator, and almost certainly will not reproduce the outputs at training points, although this will be reflected in an appropriate estimate of the emulator uncertainty at these points, which is likely to be very small so long as a sufficient number of components are retained so that the variance of the remaining eigenvalues is very small.

Note that this nugget is added to the *covariance between inputs*, meaning the total nugget effect is proportional to the between-output covariance Σ and the actual nugget ν. These are convenient assumptions for computational reasons, which ensure that:

1. the covariance function remains separable (which is not the case if the nugget is added to the overall covariance function),
2. we can still integrate out Σ (this is made possible by having the nugget proportional to Σ).

In an ideal world we might not want to *entangle* the nugget and the between outputs variance matrix Σ in this way, however without this modelling choice, that should be validated, we lose the above simplifications which can be very important in high dimensional output settings.

8.73.6 Posterior process

Assuming a *weak prior* on (β, Σ), it follows from the procedure page for building a multivariate GP emulator (*ProcBuildMultiOutputGP*) that we can analytically integrate out β and Σ, yielding a posterior t-process. Integrating out these parameters is very advantageous in this setting since these represent a rather large number of parameters which might otherwise need to be optimised or estimated.

Conditional on the covariance function length scales δ, the mean and covariance functions of the posterior t-process are given by:

$$m^*(x) = h(x)^{\rm T}\widehat{\beta} + t(x)^{\rm T}A^{-1}(f(D) - H\widehat{\beta}),$$

$$v^*(x,x') = \widehat{\Sigma}\left\{c(x,x') - t(x)^{\rm T}A^{-1}t(x') + R(x)\left(H^{\rm T}A^{-1}H\right)^{-1}R(x')^{\rm T}\right\}$$

where

- $R(x) = h(x)^{\rm T} - t(x)^{\rm T}A^{-1}H$,
- $H = h(D)^{\rm T}$,

_ **:math:‘widehat{beta}=left(H^{rm T} A^{-1} Hright)^{-1}H^{rm T} A^{-1}** f(D)‘

and

- **:math:‘widehatSigma = (n-q)^{-1} f(D)^{rm T}left{A^{-1} - A^{-1}** Hleft(H^{rm T} A^{-1} Hright)^{-1}H^{rm T}A^{-1}right} f(D)‘.

8.73.7 Parameter estimation

In order to make predictions, we need to be able to sample from the posterior distribution of δ, which for a prior $\pi_\delta(\delta)$ is given by:

$$\pi^*_\delta(\delta) \propto \pi_\delta(\delta) \times |\widehat{\Sigma}|^{(n-q)/2}|A|^{-1/2}|H^{\rm T}A^{-1}H|^{-1/2}.$$

A common approximation consists in choosing a single, optimal value for δ, typically the mode of its posterior distribution (maximum a posteriori). Alternately, if we place an improper uniform prior on δ, we can set it to the value which maximises the restricted (marginal) likelihood of the data having integrated out β and Σ, which is what is done here:

$$p(f(D)|\delta) \propto |\widehat{\Sigma}|^{(n-q)/2}|A|^{-1/2}|H^{\rm T}A^{-1}H|^{-1/2}.$$

The optimal values of δ and ν are obtained through minimisation of the negative log of the expression above using a gradient descent method. The optimisation is carried out in log-space to constrain the length scales and nugget variance to remain positive. Thus in this example we fix the nugget and length scales to their most likely values (in a restricted marginal likelihood sense) rather than integrating them out, as might be ideally done - see *ProcSimulation-BasedInference*. This reduces the computational complexity significantly, but could cause some underestimation of emulator uncertainty. Again validation suggests this effect is minimal in this case, but this should always be checked.

8.73.8 Prediction using the emulator

Using the value for δ obtained from the previous step, we compute the posterior mean $m^*(x)$ and variance $v^*(x)$ at a new input x as shown above. Further details, such as if we sample from the posterior over δ can be found in the procedure page for predicting simulator outputs using a GP emulator (*ProcPredictGP*).

8.73.9 Projecting back to real space

Since our emulator maps the (normalised) inputs to the reduced output space spanned by the first r^- principal components, we need to project the r^--dimension output of the emulator back onto the full r-dimension simulator output space. We apply the back-projection matrix P^{T} to the posterior process predictions. That is, at a given input x with predictive mean and variance $m^*(x)$ and $v^*(x)$, we can write:

$$m(x') = m^*(x)P^{\mathrm{T}}$$
$$v(x, x') = Pv^*(x, x')P^{\mathrm{T}} + \xi$$

where ξ is additional white noise to account for the fact that we have discarded some the principal components when projecting onto the r^- retained components.

8.73.10 Validation

In order to assess the quality of the emulator, we create a randomly selected test set of $n' = 300$ inputs D', at which we compute the emulator's response. We also compute the true simulator's response and compare the emulator and the simulator using the diagnostic tools detailed in the procedure page on validating a Gaussian process emulator (*ProcValidateCoreGP*).

Validation in original (log concentration) space

Figure 2 shows the (marginal) standardised errors at each test point x' (in no particular order) for the first 16 outputs. The standardised error is the error between the predictive mean $m(x')$ of the emulator and the true output $f(x')$, weighted by the predictive variance $v(x', x')$. Red lines indicate the +/-2 standard deviation intervals. The emulator does a good job (from a statistical point of view) as shown by the appropriate number of errors falling outside the +/-2 standard deviation interval, although there does appear to be something of a bias toward lower valued residuals, which we believe is related to the use of the log transform for very low concentrations.

In high dimension, visualising marginalised, standardised errors plots quickly becomes impractical. Figure 3 attempts to give a summary of the marginal distribution of the error for all 39 outputs. For each output, we compute a histogram of absolute standardised errors with bin size of 1 standard deviation. We then normalise and plot the histogram as a percentage. For instance, about 50% of the errors fall within 1 standard deviation and about 90% within 2 standard deviations. This confirms that the emulator is correctly representing the uncertainty in the prediction, but these plots are sensitive to small sample errors above 2 standard deviations, and for many outputs the emulator seems somewhat overconfident, which might be due to the fact that the length scale and nugget hyperparameters are not integrated out, but optimised and treated as known.

Further diagnostics include the Root Mean Square Error (RMSE) and Mahalanobis distance, values for which are provided in Table 1. The RMSE is a measure of the error at each test point and tells us, irrespective of the uncertainty, how close the emulator's response is to the simulator's. For the current emulator, we obtain a RMSE of 5.4 for an overall process variance (on average, at a point) of 129.4, which indicates a good fit to the test data. This can be seen to relate to the resolution of the prediction - how close it is to the right value, without reference to the predicted uncertainty. Another measure, which takes into account the uncertainty of the emulator, is the Mahalanobis distance, which for a multi-output emulator is given by:

$$M(D') = \mathrm{vec}\left[f(D') - m(D')\right]^{\mathrm{T}} C^{-1} \mathrm{vec}\left[f(D') - m(D')\right]$$

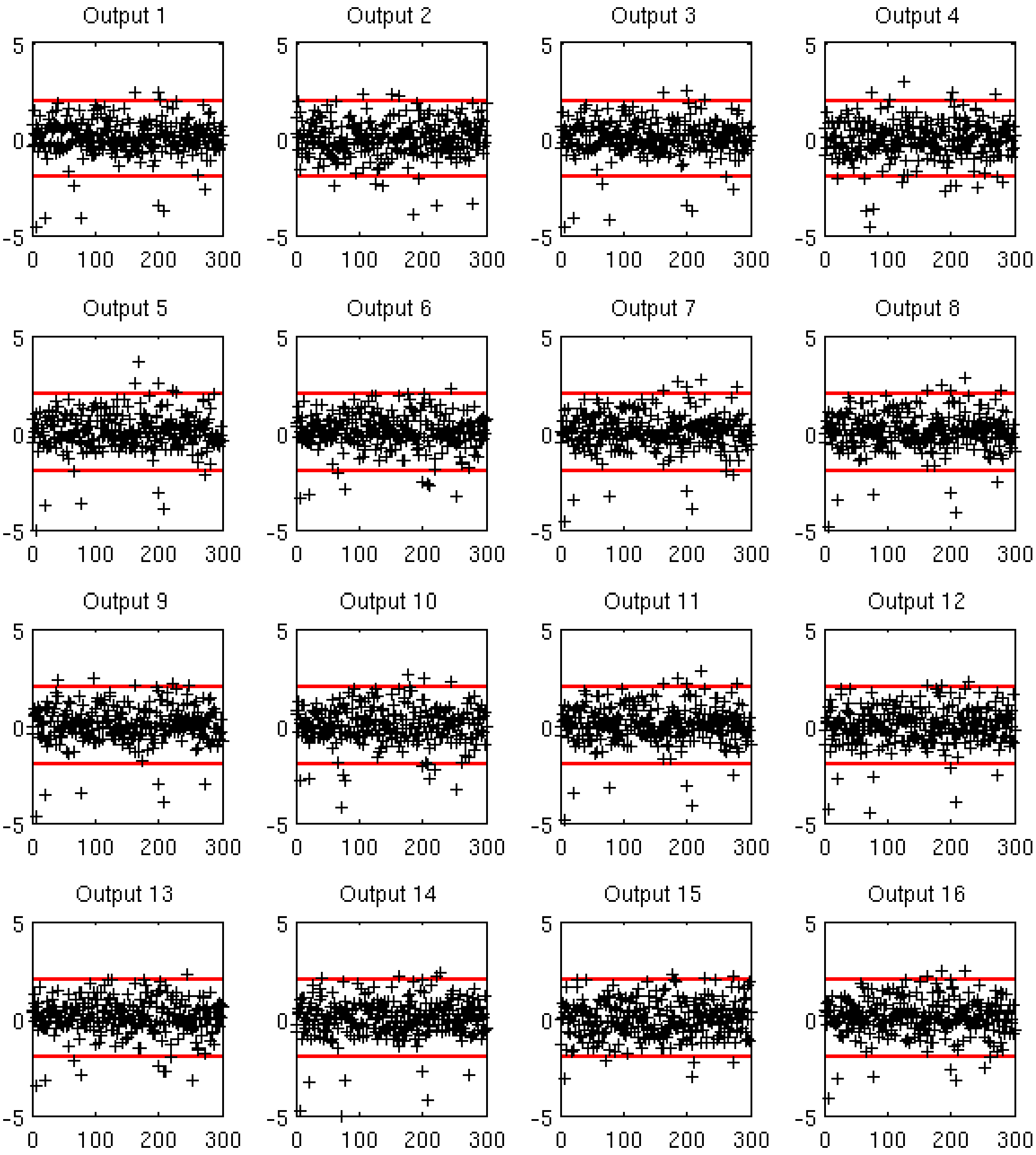


Fig. 28: **Figure 2:** Standardised errors for the 300 test points

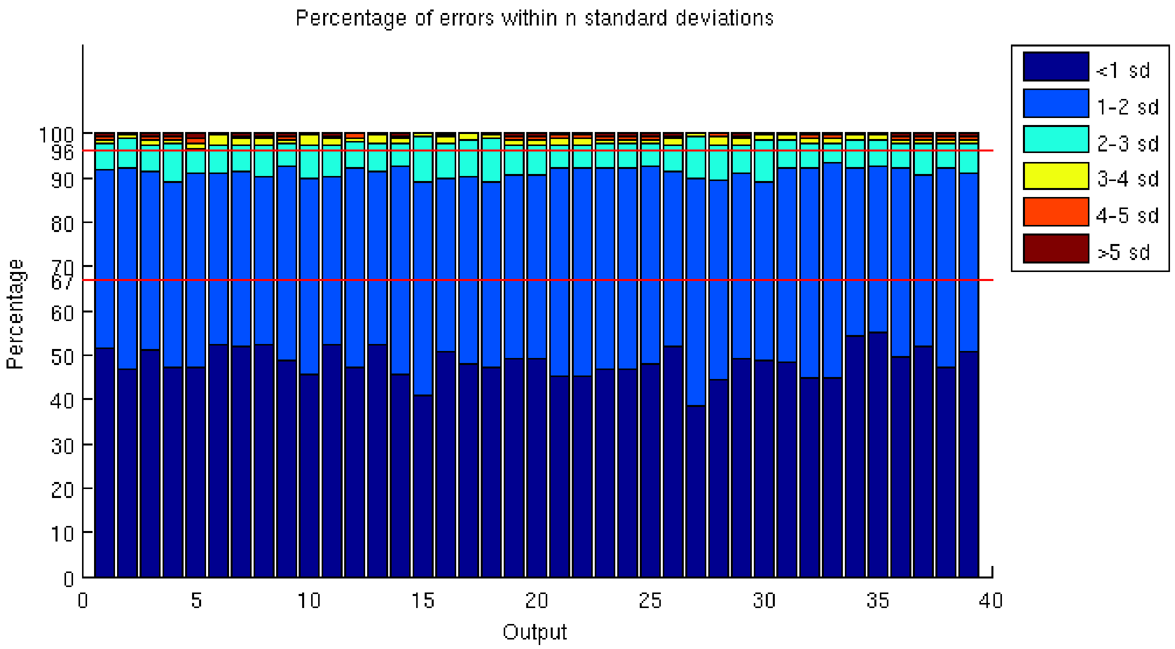


Fig. 29: **Figure 3:** Marginal distribution of absolute errors (percentages, bin size = 1 standard deviation)

where

- $\mathrm{vec}\left[\cdot\right]$ is the column-wise "vectorise" operator,
- C is the $n'r \times n'r$ covariance matrix between any two points at any two outputs, i.e. $C = \Sigma \otimes c(D', D')$.

for which the theoretical mean value is the number of test points. The theoretical mean for the Mahalanobis distance is the number of points, times the number of outputs (due to the vectorisation of the error), i.e. $E[M] = n'r$. We find that $M(D') = 11,141$ which is very close to the theoretical 11,700 and confirms the emulator is modelling the uncertainty correctly, that is the predictions are statistically reliable.

RMSE	Process amplitude	Mahalanobis distance	Theoretical mean
5.4	129.4	11,141	300x39 = 11,700

8.73.11 Discussion

This example discussed the emulation of a relatively simple chemical model, which is rather fast to evaluate. The model is quite simple, in that the response to most inputs is quite close to linear and there is significant correlation between outputs. Nevertheless it provides a god case study to illustrate some of the issues one must face in building multivariate emulators. These include

1. The size of the training set needs to be large enough to identify the parameters in the covariance function (integrating them all out is computationally very expensive), and to reasonably fill the large input space.
2. Using a separable model for the covariance produces significant simplifications allowing many parameters to be integrated out and speeding up the estimation of parameters in the emulator greatly. However there is a price, that the outputs necessarily have the same response to the inputs (modelled by the between inputs covariance function) scaled by the between outputs variance matrix Σ.
3. When using dimension reduction as part of the emulation process, care must be taken in selecting which components to retain, and careful validation is necessary to establish the quality of the emulator.

Overall the example shows that it is possible to emulate effectively in high dimensions, with the above caveats. In many cases consideration should be given as to whether a joint emulation of the outputs is really needed, but this will be addressed in another example later.

8.73.12 See also

- *ThreadVariantMultipleOutputs* (and links therein)
- *ProcBuildMultiOutputGP*
- *AltMultipleOutputsApproach*
- *ProcOutputsPrincipalComponents*
- *MetaNotation*

8.74 Example: Using Automatic Relevance Determination (ARD)

In this page we demonstrate the implementation of the ARD (see the procedure page *ProcAutomaticRelevanceDetermination*) method on a simple 1D synthetic example.

The simulator function is $f(x_1, x_2) = sin(x_1/10) + 0 \times x_2$, i.e. a two variable function which ignores the second input altogether.

A 7 point design was used to train a emulator with a squared exponential function:

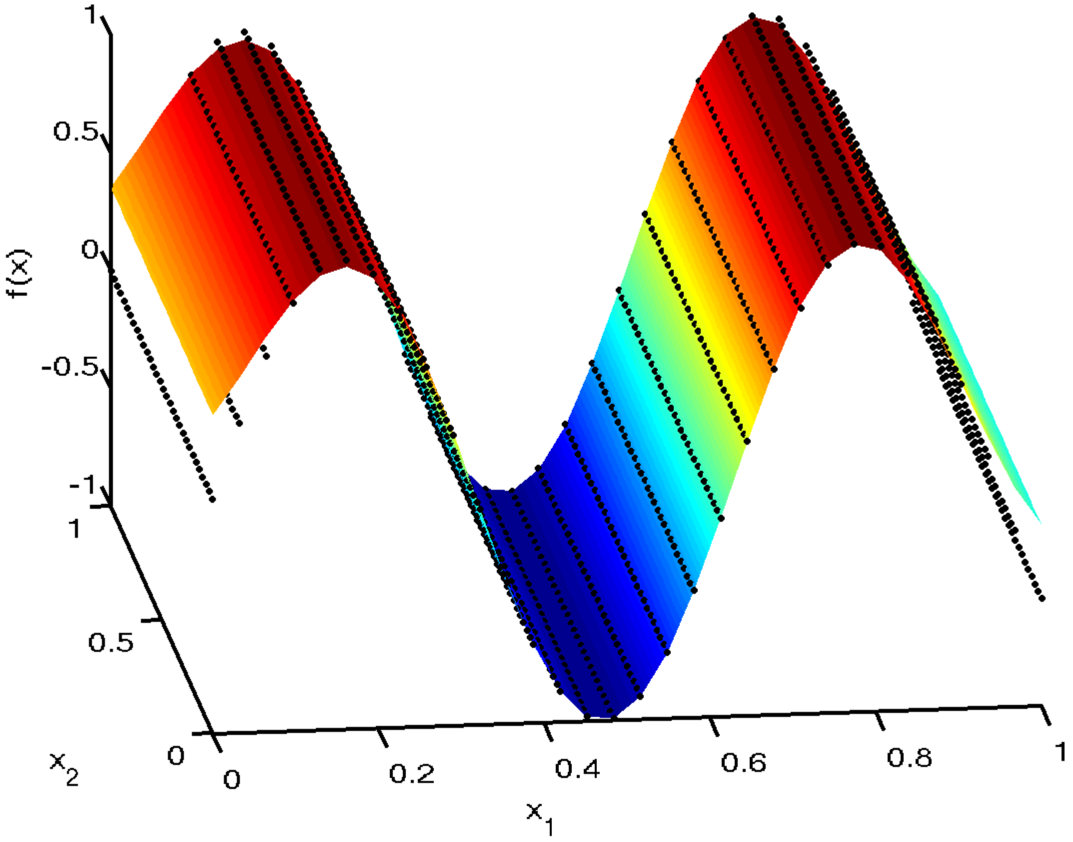
Table of Data

Variable	Value						
x_1	0.1000	0.2333	0.3667	0.5000	0.6333	0.7667	0.9000
x_2	0.2433	0.3767	0.9100	0.6433	0.1100	0.5100	0.7767
$f(x_1, x_2)$	0.8415	0.7231	-0.5013	-0.9589	0.0501	0.9825	0.4121

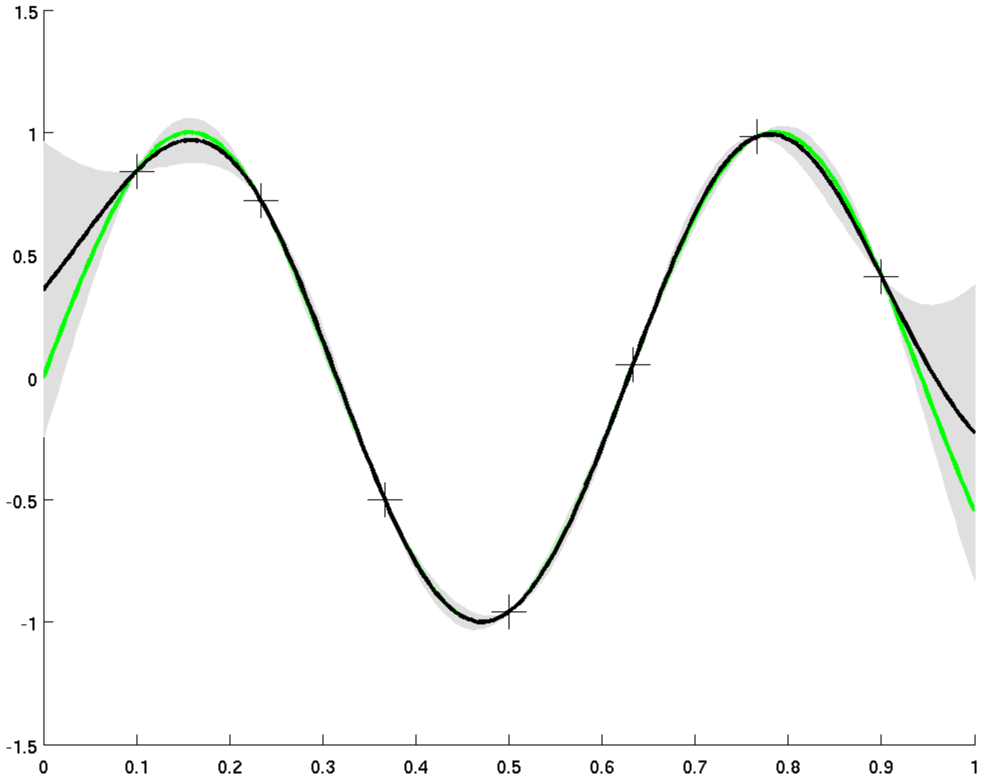
Note that both input factors are operating on the same scale so no standardisation is needed in this case.

The inference is done by using a scaled conjugate gradient algorithm to maximise the log likelihood of the Gaussian Process emulator (that is in this case no priors are placed over the length scales).

To check the fit of the emulator we used a grid test set of 1000 points. The simulator values are plotted in black dots and the emulation mean prediction is the smooth coloured surface. We can clearly see from the plot that the both the simulator and emulator responses are insensitive to the value of x_2.



To further validate the emulator and examine the output predictive variance, we plot below a profile of the simulator function at $x_2 = 1$ and x_1 a grid design of 1000 points. The simulator function is shown in **green** against the emulator prediction in **black** with the predictive variance in **grey**. The training data are also shown as crosses (although the x_2 coordinate varies in the training data but clearly this has no effect on the output value from the simulator).



The length scales obtained through maximum likelihood are $\delta_1 = 0.16$ and $\delta_2 = 48.6$ which can be interpreted as the emulator using the first variable and ignoring the second.

8.75 Example: Using the Morris method

In the page we provide an example of applying the Morris screening method (*ProcMorris*) on a simple synthetic example.

Suppose the simulator function is deterministic and described by the formula $f(x) = x_1 + x_2^2 + x_2 \times sin(x_3) + 0 \times x_4$. We will use the standard Morris method to discover the three relevant variables from a total set of four input variables.

We use $R = 4$ trajectories, four levels for each input $p = 4$ and $\Delta = p/2(p-1) = 0.66$. Given these parameters, the total number of simulator runs is $(k+1)R = 20$. The experimental design used is shown below. The values have been rounded to two decimal places.

Morris Experimental Design

x_1	x_2	x_3	x_4	$f(x)$
0.00	0.33	0.33	0.00	0.22
0.67	0.33	0.33	0.00	0.89
0.67	1.00	0.33	0.00	1.99
0.67	1.00	0.33	0.67	1.99
0.67	1.00	1.00	0.67	2.51
0.33	0.00	0.33	0.33	0.33
0.33	0.00	1.00	0.33	0.33
1.00	0.00	1.00	0.33	1.00
1.00	0.67	1.00	0.33	2.01
1.00	0.67	1.00	1.00	2.01
0.00	0.00	0.00	0.33	0.00
0.00	0.00	0.67	0.33	0.00
0.67	0.00	0.67	0.33	0.67
0.67	0.67	0.67	0.33	1.52
0.67	0.67	0.67	1.00	1.52
0.33	0.33	0.00	0.00	0.44
0.33	0.33	0.67	0.00	0.65
1.00	0.33	0.67	0.00	1.32
1.00	0.33	0.67	0.67	1.32
1.00	1.00	0.67	0.67	2.62

Using this design we sampling statistics of the elementary effects for each factor are computed:

Morris Method Indexes

Factor	μ	μ_*	σ
x_1	1	1	2e-16
x_2	1.6	1.6	0.28
x_3	0.27	0.27	0.36
x_4	0	0	0

The μ and σ values are plotted in Figure 1 below. As can be seen the Morris method effectively and clearly identifies the relevant inputs. For factor x_1 we note the high μ and low σ values signify a linear effect. For factors x_2, x_3 the large σ value demonstrates the non-linear/interaction effects. Factor x_4 has zero value for both metrics as expected for an irrelevant factor. Lastly the agreement of μ to μ_* for all factors shows a lack of cancellation effects, due to the monotonic nature of the input-output response in this simple example. In general models this will not be the case, particularly those with non-linear responses.

8.75.1 References

A multitude of screening and sensitivity analysis methods including the Morris method are implemented in the sensitivity package available for the R statistical software system:

Gilles Pujol and Bertrand Iooss (2008). Sensitivity Analysis package: A collection of functions for factor screening and global sensitivity analysis of model output. http://cran.r-project.org/web/packages/sensitivity/index.html

R Development Core Team (2005). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, http://www.R-project.org.

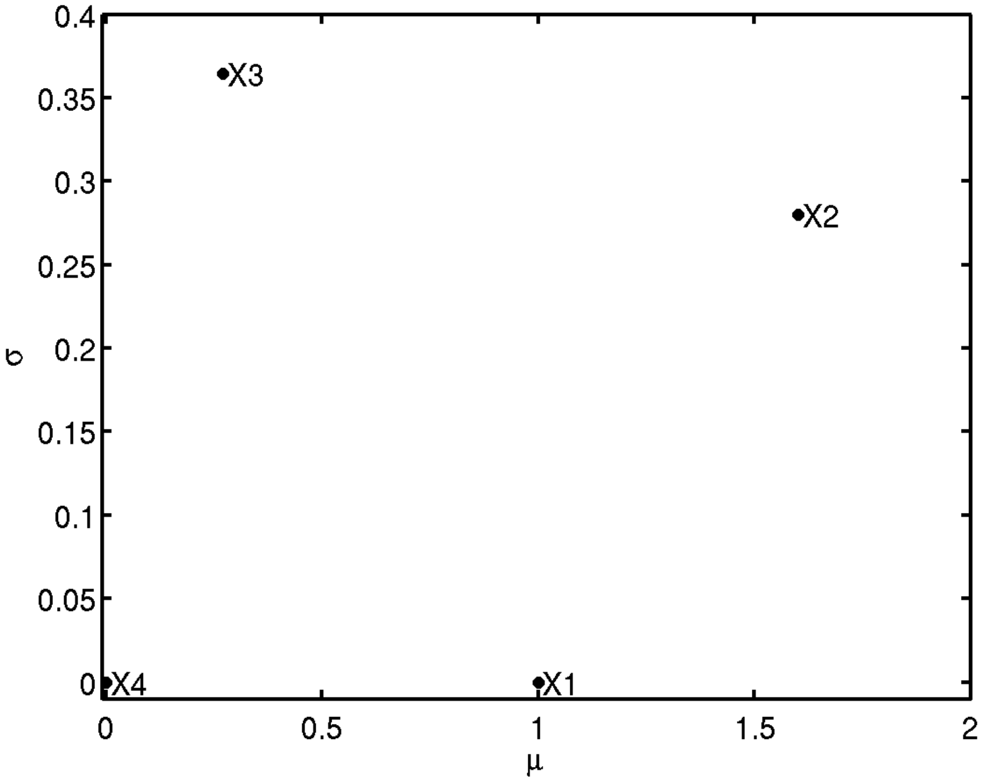


Fig. 30: **Figure 1:** Summary statistics for Morris method.

8.76 Example: A one dimensional emulator built with function output and derivatives

8.76.1 Simulator description

In this page we present an example of fitting an emulator to a $p = 1$ dimensional simulator when in addition to learning the function output of the simulator, we also learn the partial derivatives of the output w.r.t the input. The simulator we use is the function, $\sin(2x) + (\frac{x}{2})^2$. Although this is not a complex simulator which takes an appreciable amount of computing time to execute, it is still appropriate to use as an example. We restrict the range of the input, x, to lie in $[-5, 5]$ and the behaviour of our simulator over this range is shown in Figure 1. The simulator can be analytically differentiated to provide the relevant derivatives: $2\cos(2x) + \frac{x}{2}$. For the purpose of this example we define the adjoint as the function which when executed returns both the simulator output and the derivative of the simulator output w.r.t the simulator input.

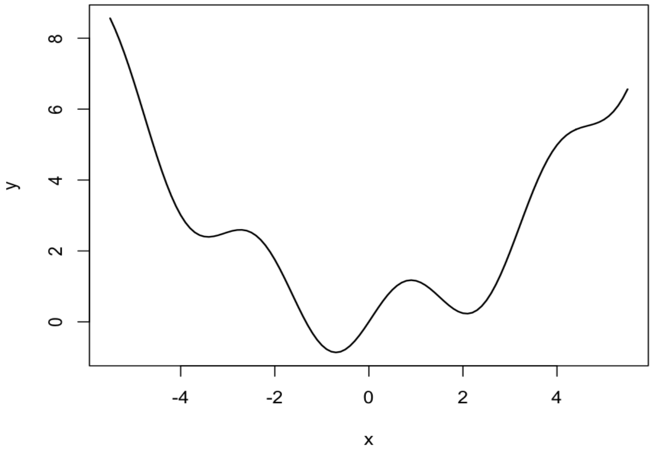


Fig. 31: **Figure 1:** The simulator over the specified region

8.76.2 Design

We need to choose a design to select at which input points the simulator is to be run and at which points we want to include derivative information. An Optimised Latin Hypercube, which for one dimension is a set of equidistant points on the space of the input variable, is chosen. We select the following 5 design points:

$$D = [-5, -2.5, 0, 2.5, 5].$$

We choose to evaluate the function output and the derivative at each of the 5 points and scale our design points to lie in $[0, 1]$. This information is added to D resulting in the design, $\tilde{D}$ of length $\tilde{n}$. A point in $\tilde{D}$ has the form (x, d), where d denotes whether a derivative or the function output is to be included at that point; for example $(x, d) = (x, 0)$ denotes the function output at point x and $(x, d) = (x, 1)$ denotes the derivative w.r.t to the first input at point x.

This results in the following design:

$$\begin{aligned}\tilde{D} &= [(x_1, d_1), (x_2, d_2), \cdots, (x_{10}, d_{10})]\\ &= [(0, 0), (0.25, 0), (0.5, 0), (0.75, 0), (1, 0), (0, 1), (0.25, 1), (0.5, 1), (0.75, 1), (1, 1)]\end{aligned}$$

The output of the adjoint at these points, which make up our training data, is:

$$\tilde{f}(\tilde{D}) = [6.794, 2.521, 0, 0.604, 5.706, -41.78, -6.827, 20, 18.17, 8.219]^{\mathrm{T}}.$$

Note that the first 5 elements of $\tilde{D}$, and therefore $\tilde{f}(\tilde{D})$, are simply the same as in the core problem. Elements 6-10 correspond to the derivatives. Throughout this example n will refer to the number of distinct locations in the design (5), while $\tilde{n}$ refers to the total amount of data points in the training sample. Since we include a derivative at all n points w.r.t to the $p = 1$ input, $\tilde{n} = n(p+1) = 10$.

8.76.3 Gaussian process setup

In setting up the Gaussian process, we need to define the mean and the covariance function. As we are including derivative information here we need to ensure that the mean function is once differentiable and the covariance function is twice differentiable. For the mean function, we choose the linear form described in the alternatives page on emulator prior mean function (*AltMeanFunction*), which is $h(x) = [1, x]^{\mathrm{T}}$ and $q = 1 + p = 2$. This corresponds to $\tilde{h}(x, 0) = [1, x]^{\mathrm{T}}$ and we also have $\frac{\partial}{\partial x} h(x) = [1]$ and so $\tilde{h}(x, 1) = [1]$.

For the covariance function we choose $\sigma^2 c(\cdot, \cdot)$, where the correlation function $c(\cdot, \cdot)$ has the Gaussian form described in the alternatives page on emulator prior correlation function (*AltCorrelationFunction*). We can break this down into 3 cases, as described in the procedure page for building a GP emulator with derivative information (*ProcBuildWithDerivsGP*):

Case 1 is the correlation between points, so we have $d_i = d_j = 0$ and $c\{(x_i, 0), (x_j, 0)\} = \exp\left[-\{(x_i - x_j)/\delta\}^2\right]$.

Case 2 is the correlation between a point, x_i, and the derivatives at point x_j, so we have $d_i = 0$ and $d_j \neq 0$. Since in this example $p = 1$, this amounts to $d_j = 1$ and we have:

$$\frac{\partial}{\partial x_j} c\{(x_i, 0), (x_j, 1)\} = \frac{2}{\delta^2}\left(x_i - x_j\right) \exp\left[-\{(x_i - x_j)/\delta\}^2\right].$$

Case 3 is the correlation between two derivatives at points x_i and x_j. Since in this example $p = 1$, the only relevant correlation here is when $d_i = d_j = 1$ (which corresponds to Case 3a in *ProcBuildWithDerivsGP*) and is given by:

$$\frac{\partial^2}{\partial x_i \partial x_j} c\{(x_i, 1), (x_j, 1)\} = \left(\frac{2}{\delta^2} - \frac{4\left(x_i - x_j\right)^2}{\delta^4}\right) \exp\left[-\{(x_i - x_j)/\delta\}^2\right].$$

Each Case provides sub-matrices of correlations and we arrange them as follows:

$$\tilde{A} = \begin{pmatrix} \text{Case 1} & \text{Case 2} \\ \text{Case 2} & \text{Case 3} \end{pmatrix},$$

an $\tilde{n} \times \tilde{n}$ matrix. The matrix $\tilde{A}$ is symmetric and within $\tilde{A}$ we have symmetric sub-matrices, Case 1 and Case 3. Case 1 is an $n \times n = 5 \times 5$ matrix and is exactly the same as in the procedure page *ProcBuildCoreGP*. Since we are including the derivative at each of the 5 design points, Case 2 and 3 sub-matrices are also of size $n \times n = 5 \times 5$.

8.76.4 Estimation of the correlation length

We need to estimate the correlation length δ. In this example we will use the value of δ that maximises the posterior distribution $\pi^*_\delta(\delta)$, assuming that there is no prior information on δ, i.e. $\pi(\delta) \propto \mathrm{const}$. The expression that needs to be maximised is (from *ProcBuildWithDerivsGP*)

$$\pi^*_\delta(\delta) \propto (\widehat{\sigma}^2)^{-(\tilde{n}-q)/2} |\tilde{A}|^{-1/2} |\tilde{H}^{\mathrm{T}} \tilde{A}^{-1} \tilde{H}|^{-1/2}.$$

where

$$\widehat{\sigma}^2 = (\tilde{n} - q - 2)^{-1} \tilde{f}(\tilde{D})^{\mathrm{T}} \left\{\tilde{A}^{-1} - \tilde{A}^{-1}\tilde{H}\left(\tilde{H}^{\mathrm{T}}\tilde{A}^{-1}\tilde{H}\right)^{-1}\tilde{H}^{\mathrm{T}}\tilde{A}^{-1}\right\} \tilde{f}(\tilde{D}).$$

We have $\tilde{H} = [\tilde{h}(x_1, d_1), \ldots, \tilde{h}(x_{10}, d_{10})]^{\mathrm{T}}$, where $\tilde{h}(x, d)$ and $\tilde{A}$ are defined above in section Gaussian process setup.

Recall that in the above expressions the only term that is a function of δ is the correlation matrix $\tilde{A}$.

The maximum can be obtained with any maximisation algorithm and in this example we used Nelder - Mead. The value of δ which maximises the posterior is 0.183 and we will fix δ at this value thus ignoring the uncertainty with it, as discussed in *ProcBuildCoreGP*. We refer to this value of δ as $\hat{\delta}$. We have scaled the input to lie in $[0, 1]$ and so in terms of the original input scale, $\hat{\delta}$ corresponds to a smoothness parameter of $10 \times 0.183 = 1.83$

8.76.5 Estimates for the remaining parameters

The remaining parameters of the Gaussian process are β and σ^2. We assume weak prior information on β and σ^2 and so having estimated the correlation length, the estimate for σ^2 is given by the equation above in section Estimation of the correlation length, and the estimate for β is

$$\hat{\beta} = \left(\tilde{H}^{\mathrm{T}}\tilde{A}^{-1}\tilde{H}\right)^{-1}\tilde{H}^{\mathrm{T}}\tilde{A}^{-1}\tilde{f}(\tilde{D}).$$

Note that in these equations, the matrix $\tilde{A}$ is calculated using $\hat{\delta}$. The application of the two equations for $\hat{\beta}$ and $\widehat{\sigma}^2$, gives us in this example $\hat{\beta} = [4.734, -2.046]^{\mathrm{T}}$ and $\widehat{\sigma}^2 = 15.47$

8.76.6 Posterior mean and Covariance functions

The expressions for the posterior mean and covariance functions as given in *ProcBuildWithDerivsGP* are

$$m^*(x) = h(x)^{\mathrm{T}}\widehat{\beta} + \tilde{t}(x)^{\mathrm{T}}\tilde{A}^{-1}(\tilde{f}(\tilde{D}) - \tilde{H}\widehat{\beta})$$

and

$$v^*(x_i, x_j) = \widehat{\sigma}^2\{c(x_i, x_j) - \tilde{t}(x_i)^{\mathrm{T}}\tilde{A}^{-1}\tilde{t}(x_j) + \left(h(x_i)^{\mathrm{T}} - \tilde{t}(x_i)^{\mathrm{T}}\tilde{A}^{-1}\tilde{H}\right)\left(\tilde{H}^{\mathrm{T}}\tilde{A}^{-1}\tilde{H}\right)^{-1}\left(h(x_j)^{\mathrm{T}} - \tilde{t}(x_j)^{\mathrm{T}}\tilde{A}^{-1}\tilde{H}\right)^{\mathrm{T}}\}.$$

Figure 2 shows the predictions of the emulator for 100 points uniformly spaced on the original scale. The solid, black line is the output of the simulator and the blue, dashed line is the emulator mean m^* evaluated at each of the 100 points. The blue dotted lines represent 2 times the standard deviation about the emulator mean, which is the square root of the diagonal of matrix v^*. The black crosses show the location of the design points where we have evaluated the function output and the derivative to make up the training sample. The green circles show the location of the validation data which is discussed in the section below. We can see from Figure 2 that the emulator mean is very close to the true simulator output and the uncertainty decreases as we get closer the location of the design points.

8.76.7 Validation

In this section we validate the above emulator according to the procedure page for validating a GP emulator (*ProcValidateCoreGP*).

The first step is to select the validation design. We choose here 15 space filling points ensuring these points are distinct from the design points. The validation points are shown by green circles in Figure 2 above and in the original input space of the simulator are:

$$[-4.8, -4.3, -3.6, -2.9, -2.2, -1.5, -0.8, -0.1, 0.6, 1.3, 2, 2.7, 3.4, 4.1, 4.8]$$

and in the transformed space:

$$D' = [x'_1, x'_2, \cdots, x'_{15}] = [0.02, 0.07, 0.14, 0.21, 0.28, 0.35, 0.42, 0.49, 0.56, 0.63, 0.7, 0.77, 0.84, 0.91, 0.98].$$

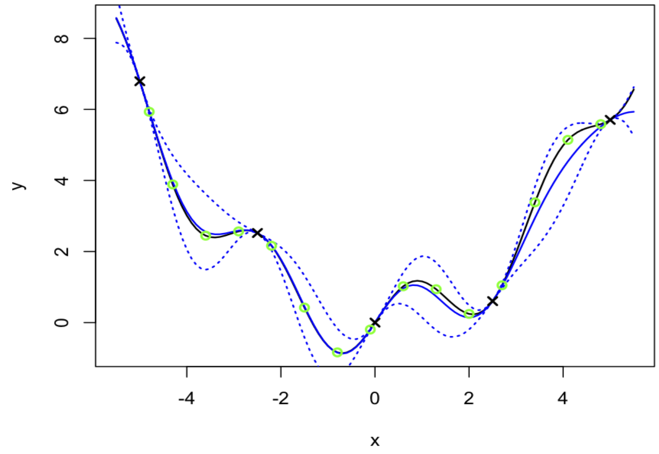


Fig. 32: **Figure 2:** The simulator (solid black line), the emulator mean (blue, dotted) and 95% confidence intervals shown by the blue, dashed line. Black crosses are design points, green circles are validation points.

Note that the prime symbol $'$ does not denote a derivative, as in *ProcValidateCoreGP* we use the prime symbol to specify validation. We're predicting function output in this example and so do not need validation derivatives; as such we have a validation design D' and not $\tilde{D}'$. The function output of the simulator at these validation points is

$$f(D') = [5.934, 3.888, 2.446, 2.567, 2.161, 0.421, -0.840, -0.196, 0.102, 0.938, 0.243, 1.050, 3.384, 5.143, 5.586]^{\mathrm{T}}$$

We then calculate the mean $m^*(\cdot)$ and variance $v^*(\cdot,\cdot)$ of the emulator at each validation design point in D' and the difference between the emulator mean and the simulator output at these points can be compared in Figure 2.

We also calculate standardised errors given in *ProcValidateCoreGP* as $\frac{f(x_j')-m^*(x_j')}{\sqrt{v^*(x_j',x_j')}}$ and plot them in Figure 3.

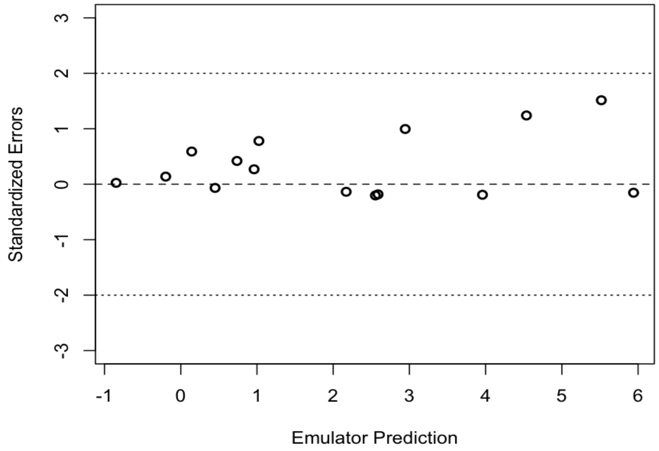


Fig. 33: **Figure 3:** Individual standardised errors for the prediction at the validation points

Figure 3 shows that all the standardised errors lie between -2 and 2 providing no evidence of conflict between simulator and emulator.

We calculate the Mahalanobis distance as given in *ProcValidateCoreGP*:

$$M = (f(D') - m^*(D'))^{\rm T}(v^*(D',D'))^{-1}(f(D') - m^*(D')) = 6.30$$

when its theoretical mean is $\mathrm{E}[M] = n' = 15$ and variance, $\mathrm{Var}[M] = \frac{2n'(n'+\tilde{n}-q-2)}{\tilde{n}-q-4} = 12.55^2$.

We have a slightly small value for the Mahalanobis distance therefore, but it is within one standard deviation of the theoretical mean. The validation sample is small and so we would only expect to detect large problems with this test. This is just an example and we would not expect a simulator of a real problem to only have one input, but with our example we can afford to run the simulator intensely over the specified input region. This allows us to assess the overall performance of the emulator and, as Figure 2 shows, the emulator can be declared as valid.

8.76.8 Comparison with an emulator built with function output alone

We now build an emulator for the same simulator with all the same assumptions, but this time leave out the derivative information to investigate the effect of the derivatives and compare the results.

We obtain the following estimates for the parameters:

$$\hat{\delta} = 2.537$$
$$\hat{\beta} = [82.06, 54.34]^{\rm T}$$
$$\widehat{\sigma}^2 = 49615.$$

Figure 4 shows the predictions of this emulator for 100 points uniformly spaced on the original scale. The solid, black line is the output of the simulator and the red, dashed line is the emulator mean evaluated at each of the 100 points. The red dotted lines represent 2 times the standard deviation about the emulator mean. The black crosses show the location of the design points where we have evaluated the function output. We can see from Figure 4 that the emulator is not capturing the behaviour of the simulator at all and further simulator runs are required.

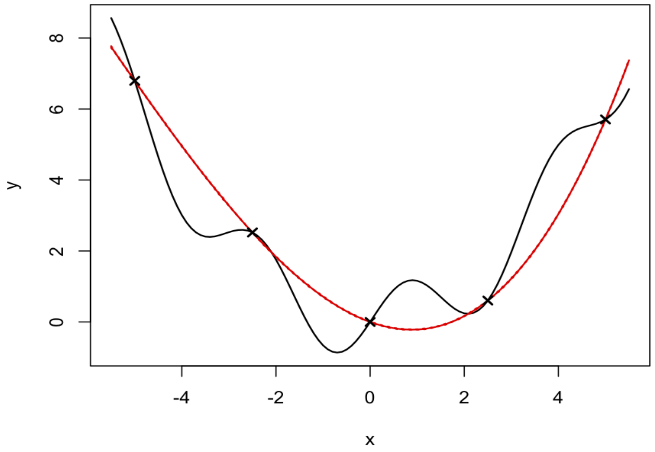


Fig. 34: **Figure 4:** The simulator (solid black line), the emulator mean (red, dotted) and 95% confidence intervals shown by the red, dashed line. Black crosses are design points.

We add 4 further design points, $[-3.75, -1.25, 1.25, 3.75]$, and rebuild the emulator, without derivatives as before. This results in new estimates for the parameters, $\hat{\delta} = 0.177$, $\hat{\beta} = [4.32, -2.07]^{\rm T}$ and $\widehat{\sigma}^2 = 15.44$, and Figure 5 shows the predictions of this emulator for the same 100 points. We now see that the emulator mean closely matches the simulator output across the specified range.

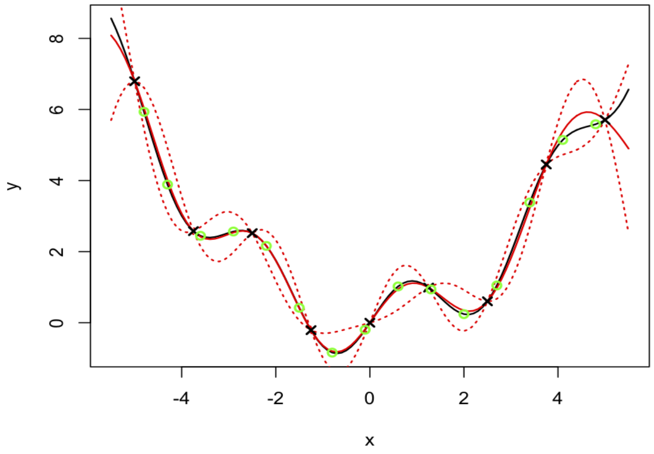


Fig. 35: **Figure 5:** The simulator (solid black line), the emulator mean (red, dotted) and 95% confidence intervals shown by the red, dashed line. Black crosses are design points, green circles are validation points.

We repeat the validation diagnostics using the same validation data and obtain a Mahalanobis distance of 4.70, while the theoretical mean is 15 with standard deviation 14.14. As for the emulator with derivatives, a value of 4.70 is bit small; however the standardised errors calculated as before, and shown in Figure 6 below, provide no evidence of conflict between simulator and emulator and the overall performance of the emulator as illustrated in Figure 5 is satisfactory.

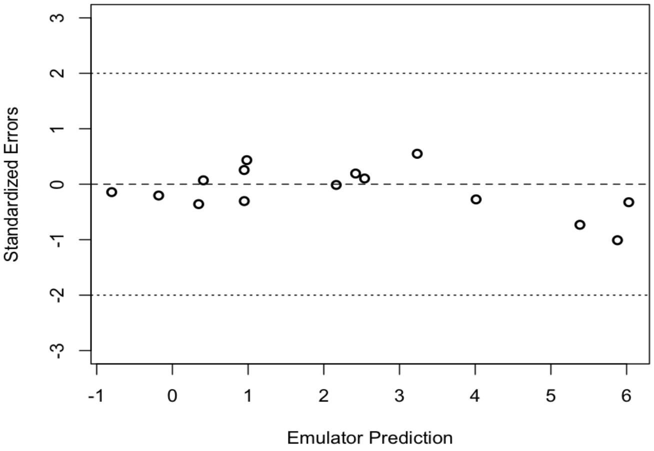


Fig. 36: **Figure 6:** Individual standardised errors for the prediction at the validation points

We have therefore now built a valid emulator without derivatives but required 4 extra simulator runs to the emulator with derivatives, to achieve this.

8.77 Discussion: Active and inactive inputs

8.77.1 Description and Background

Simulators usually have many inputs. When we wish to build an *emulator* to represent the simulator over some region of the input space, it is desirable to minimise the dimensionality of that space (i.e. the number of inputs). This dimension reduction is motivated by two considerations.

First, decreasing the number of inputs decreases the complexity of the emulation, and in particular reduces the number of simulator runs that we need in the *training sample*. If the number of inputs is large, for instance more than 30, emulation may become impractical. Difficulty may be experienced in estimating *hyperparameters* (because there are now many hyperparameters to estimate) and in doing other computations (because the large training sample leads to large, possibly ill-conditioned, matrix inversions).

The second motivation is that we often find in practice that most of the inputs have only small influences on the output(s) of interest. If, when we vary an input over the range for which we wish to build the emulator, the output changes only slightly, then we can identify this as an *inactive input*. If we can build the emulator using only *active inputs* then not only will the emulation task become much simpler but we will also have an emulator that represents the simulator accurately but without unnecessary complexity.

8.77.2 Discussion

Identifying active inputs

The process of identifying active inputs is called *screening* (or dimension reduction). Screening can be thought of as an application of *sensitivity analysis*, the active inputs being those which have the highest values of an appropriate sensitivity measure.

A detailed discussion of screening, with procedures for recommended methods, is available at the screening topic thread (*ThreadTopicScreening*).

Excluding inactive inputs

There are two ways to exclude inactive inputs when building and using an emulator.

Inactive inputs fixed

The first method is to fix all the inactive inputs at some specified values. The fixed values may be arbitrary, since the notion of an inactive input is that its value is unimportant. We are thereby limiting the input space to that part in which the inactive inputs take these fixed values and only the active inputs are varied.

In this approach, a *deterministic* simulator remains deterministic. It is a function only of the active inputs, and the emulator only represents the simulator in this reduced input space. Although the values of inactive inputs are unimportant, this does not mean that varying the inactive inputs has no effect at all on the output. So the emulator cannot strictly be used to predict the output of the simulator at any other values of the inactive inputs than the chosen fixed values.

Notice that the emulator is now built using a training sample in which all the inactive variables are fixed at their chosen values in every simulator run. The screening process will generally have required a number of simulator runs in which all the inputs are varied, in order to identify which are inactive. The fixed inactive inputs approach then cannot use these runs as part of its training sample, so the training sample involves a new set of simulator runs.

Inactive inputs ignored

The second method is to ignore the values of inactive inputs in training data. The values of inactive inputs are allowed to vary, and so simulator runs that were used for screening may be reusable for building the emulator. However, now the simulator output is not a deterministic function of the active inputs only.

We therefore model the output from the simulator as stochastic, equal to a function of the active inputs (which we emulate) plus a random noise term to account for the ignored values of the inactive inputs. Since the inactive inputs have only a small effect on the output, the noise term should be small, but it needs to be accounted for in the analysis.

When building the emulator, we only use the values of the active inputs. The *GP* correlation function involves an added *nugget* term as discussed in the alternatives page on emulator prior correlation function (*AltCorrelationFunction*).

Prediction also needs to allow for the inactive inputs. Using the emulator with the correlation function including the nugget term allows us to predict the simulator outputs at any input configurations. The emulator will ignore the inactive input values, but will allow for the extra noise in predicting actual outputs via the nugget term. Removing the nugget term (by setting the corresponding hyperparameter to zero) will predict average values of the output for the given values of the active inputs and averaged over the inactive inputs.

8.77.3 Additional Comments

The choice of active inputs is a trade-off between emulating the simulator as well as possible and achieving manageable and stable computations. It also depends on context. For instance, we may initially emulate the simulator using only a very small number of the most active inputs in order to explore the simulator quickly. However, this choice will entail a larger noise term (nugget) and if the emulator predictions are not sufficiently precise we can refit the emulator using more active inputs.

A similar strategy may be employed when calibrating a simulator using real-world observations. Initial coarse emulation suffices to narrow the search for plausible values of calibration inputs, but this is then iteratively refined. At each stage, we need more precise emulation, which may be achieved partly by larger training samples but may also require the use of more active inputs.

8.78 Discussion: Adjusting Exchangeable Beliefs

8.78.1 Overview

Second-order exchangeability judgements about a collection of quantities can be used to adjust our beliefs and make inference about population means. There are many methods which address this problem, which can also be extended to the adjustment of beliefs about population variances and covariances. This requires some additional consideration of the specification of our prior beliefs as well as the organisation of such an adjustment.

This concept is of particular relevance as we typically consider the residual process from an *emulator* to have a second-order exchageable form. This means that we can employ these methods to learn about the population mean and variance of the emulator residuals. This process is described in *ProcBLVarianceLearning*, whereas this page focuses on the general methodology.

This page assumes that the reader is familiar with the concepts of *exchangeability* and *second-order exchangeability*. The analysis and discussion will be from a *Bayes linear* perspective.

8.78.2 The Second-Order Representation Theorem

Suppose that we have a collection of random variables $Y = \{Y_1, Y_2, \dots\}$, which we judge to be second-order exchangeable (SOE). Second-order exchangeability induces the following belief specification for Y:

- $\mathrm{E}[Y_i] = \mu$,
- $\mathrm{Var}[Y_i] = \Sigma$,
- $\mathrm{Cov}[Y_i, Y_j] = \Gamma$,

for all $i \neq j$ where μ, Σ, and Γ are the population means, variances and covariances.

It can be shown that for an infinite, second-order exchangeable sequence Y_i can be decomposed into two components - a **mean** component and a **residual** component. The justification for this decomposition is the representation theorem for second-order exchangeable random variables (see section 6.4 of Goldstein & Woof). Application of this representation theorem imposes the following structure and belief specifications:

1. $Y_i = \mathcal{M}(Y) + \mathcal{R}_i(Y)$ - Each individual Y_i is decomposed into the population mean vector, $\mathcal{M}(Y)$, and its individual residual vector, $\mathcal{R}_i(Y)$.
2. $\mathrm{E}[\mathcal{M}(Y)] = \mu$, and $\mathrm{Var}[\mathcal{M}(Y)] = \Gamma = \mathrm{Cov}[Y_i, Y_j]$ - The expectation of the mean component is the population mean of the Y_i, and since the mean component is common to all Y_i it is the source of all the covariance between the Y_i,
3. $\mathrm{E}[\mathcal{R}_i(Y)] = 0$, $\mathrm{Var}[\mathcal{R}_i(Y)] = \Sigma - \Gamma$, and $\mathrm{Cov}[\mathcal{R}_i(Y), \mathcal{R}_j(Y)] = 0$, $\forall i \neq j$ - The residual component is individual to each Y_i and has mean 0, constant variance, and zero correlation across the Y_i – i.e. the residuals are themselves second-order exchangeable.
4. $\mathrm{Cov}[\mathcal{M}(Y), \mathcal{R}_i(Y)] = 0$ - The mean and residual components are uncorrelated.

Note here that μ, Σ, and Γ are just belief specifications - these quantities are known and fixed - however $\mathcal{M}[Y]$ and the $\mathcal{R}_j(Y)$s are uncertain and we consider learning about $\mathcal{M}(Y)$ and the variance of the $\mathcal{R}_j(Y)$s.

8.78.3 Adjusting beliefs about the mean of a collection of exchangeable random vectors

We now describe how we can use a sample of exchangeable data to adjust our beliefs about the underlying population quantities. Continuing with the notation of previous sections, our goal is to consider how our beliefs about the population mean $\mathcal{M}(Y)$ are adjusted when we observe a sample of n values of our exchangeable random quantities, $D_n = (Y_1, \ldots, Y_n)$.

It can be shown that when adjusting beliefs about $\mathcal{M}(Y)$ by the entire sample D_n, the adjustment of our beliefs by the sample mean

$$\bar{Y}_n = \frac{1}{n}\sum_{i=1}^{n} Y_i$$

is sufficient - see Section 6.10 of the Goldstein & Wooff for details. Since the Y_i are exchangeable, we can express the sample mean, from a sample of size n, as

$$\bar{Y}_n = \mathcal{M}(Y) + \bar{\mathcal{R}}_n(Y),$$

where $\bar{\mathcal{R}}_n(Y)$ is the mean of the n residuals, and so

- $\mathrm{E}[\bar{Y}_n] = \mu$,
- $\mathrm{Var}[\bar{Y}_n] = \Gamma + \frac{1}{n}(\Sigma - \Gamma)$,
- $\mathrm{Cov}[\bar{Y}_n, \mathcal{M}(Y)] = \Gamma$.

We can then apply the Bayes linear adjustment formulae directly to find our adjusted beliefs about the population mean, $\mathcal{M}(Y)$, given the mean of a sample of size n as

$$\mathrm{E}_n[\mathcal{M}(Y)] = \mu + \Gamma\left(\Gamma + \frac{1}{n}(\Sigma - \Gamma)\right)^{-1}(\bar{Y}_n - \mu),$$

with corresponding adjusted variance

$$\mathrm{Var}_n[\mathcal{M}(Y)] = \Gamma - \Gamma\left(\Gamma + \frac{1}{n}(\Sigma - \Gamma)\right)^{-1}\Gamma.$$

8.78.4 Adjusting beliefs about the variance of a collection of exchangeable random vectors

We can apply the same methodology to learn about the population variance of a collection of exchangeable random quantities. In order to do so, we must make one additional assumption (namely that the $\mathcal{R}_i(Y)^2$ are SOE in addition to $\mathcal{R}_i(Y)$), and we require specifications of our uncertainty about the variances expressed via fourth-order moments. Methods for choosing appropriate prior specifications are discussed in the next section.

Suppose that $Y_1, Y_2, \dots$ is an infinite exchangeable sequence of scalars as above, where $\mathrm{E}[Y_i] = \mu$, $\mathrm{Var}[Y_i] = \sigma^2$, and $\mathrm{Cov}[Y_i, Y_j] = \gamma$. Then we have the standard second-order exchangeability representation

$Y_i = \mathcal{M}(Y) + \mathcal{R}_i(Y)$, where the $\mathcal{R}_1(Y), \mathcal{R}_2(Y), \dots$ are SOE with variance $\mathrm{Var}[\mathcal{R}_i(Y)] = \sigma^2 - \gamma$. How we proceed from here depends on whether we can consider the population mean $\mathcal{M}(Y)$ to be known or unknown.

$\mathcal{M}(Y)$ known

In the case where the population mean, $\mathcal{M}(Y)$, is known then there is no uncertainty surrounding the value of μ. In which case we can consider $\mathrm{Var}[\mathcal{M}(Y)] = \gamma = 0$.

To learn about population variance using this methodology, we require an appropriate exchangeability representation for an appropriate quantity. Consider the $V_i = [\mathcal{R}_i(Y)]^2 = (Y_i - \mathcal{M}(Y))^2$ which are directly observable when $\mathcal{M}(Y)$ is known. If we assume that this sequence of squared residuals $V_1, V_2, \dots$ is also SOE, then we have the representation

$$V_i = [\mathcal{R}_i(Y)]^2 = \mathcal{M}(V) + \mathcal{R}_i(V),$$

where, as before, $\mathcal{M}(V)$ is the population mean of the $V_i = (Y_i - \mathcal{M}(Y))^2$ and is hence the **population variance** of the Y_i. To learn about $\mathcal{M}(V)$ (and hence to learn about the population variance of the Y_i) we require the following belief specifications:

- $\mathrm{E}[\mathcal{M}(V)] = \omega_Y = \sigma^2$ – our expectation of the population variance of the Y_i,
- $\mathrm{Var}[\mathcal{M}(V)] = \omega_{\mathcal{M}}$ – our uncertainty associated with the population variance of Y_i, which can be resolved by observation of additional data,
- $\mathrm{Var}[\mathcal{R}_i(V)] = \omega_{\mathcal{R}}$ – irresolvable "residual uncertainty" in the V_i,

and that the sequence of $\mathcal{R}_i(V)$ are uncorrelated with zero mean. We can then use this representation and belief specification to apply the adjustment described above to revise our beliefs about the population variance $\mathcal{M}(V)$. As the sample mean is sufficient for the adjustment of beliefs in these situations and is directly observable, we calculate and adjust by the sample mean of the V_i

$$\bar{V}_n = \bar{Y}_n^{(2)} = \sum_{i=1}^{n}(Y_i - \mathcal{M}(Y))^2.$$

We then evaluate our adjusted expectation and variance of the population variance to be

$$\mathrm{E}_{\bar{V}_n}[\mathcal{M}(V)] = \frac{\omega_{\mathcal{M}}\bar{V}_n + \frac{1}{n}\omega_{\mathcal{R}}\omega_Y}{\omega_{\mathcal{M}} + \frac{1}{n}\omega_{\mathcal{R}}},$$

$$\mathrm{Var}_{\bar{V}_n}[\mathcal{M}(V)] = \frac{\frac{1}{n}\omega_{\mathcal{M}}\omega_{\mathcal{R}}}{\omega_{\mathcal{M}} + \frac{1}{n}\omega_{\mathcal{R}}},$$

where $\mathrm{E}_{\bar{V}_n}[\mathcal{M}(V)]$ represents our adjusted beliefs about the population variance of the Y_i, and $\mathrm{Var}_{\bar{V}_n}[\mathcal{M}(V)]$ represents our remaining uncertainty.

$\mathcal{M}(Y)$ unknown

Suppose we have an identical setup as before, only now the population mean of the Y_i, $\mathcal{M}(Y)$, is no longer known – ie Var$[\mathcal{M}(Y)] = \gamma > 0$. Additionally, since the population mean is now uncertain the quantities $V_i = (Y_i - \mathcal{M}(Y))^2$ are no longer directly observable so we can no longer calculate and adjust by $\bar{V}_n$. Therefore, we must construct an alternative adjustment using appropriate combinations of observables which will be informative for the population variance. Suppose that we take a sample of size $n \geq 2$, and that we calculate the sample variance, s^2 in the usual way. We then obtain the following representation (see section 8.2 of Goldstein & Wooff for details):

$$s^2 = \mathcal{M}(V) + T.$$

We can then express our beliefs about s^2 as:

- E$[s^2] = \omega_Y$,
- Var$[s^2] = \omega_{\mathcal{M}} + \omega_T$,
- Cov$[s^2, \mathcal{M}(V)] = \omega_{\mathcal{M}}$,
- and Var$[T] = \omega_T = \frac{1}{n}\omega_{\mathcal{R}} + \frac{2}{n(n-1)}[\omega_{\mathcal{M}} + \omega_Y^2]$.

Thus the sample variance, s^2, can be related directly to the population variance, $\mathcal{M}(V)$, and so we can (in principle) use the directly-observable s^2 to learn about the population variance. Before we can make the adjustment, we must make some additional assumptions that the residuals have certain fourth-order uncorrelated properties - namely that for $k \neq j \neq i$, $\mathcal{R}_j(Y)\mathcal{R}_k(Y)$ is uncorrelated with $\mathcal{M}(Y)$ and $\mathcal{R}_i(Y)$, and that for $k > j, w > u$ that $\mathcal{R}_k(Y)\mathcal{R}_j(Y)$ and $\mathcal{R}_e(Y)\mathcal{R}_u(Y)$ are also uncorrelated. With these assumptions and specifications, the adjusted expectation and variance of the population variance given s^2 are given by

$$\begin{aligned} \mathrm{E}_{s^2}[\mathcal{M}(V)] &= \frac{\omega_{\mathcal{M}} s^2 + \omega_T \omega_Y}{\omega_{\mathcal{M}} + \omega_T}, \\ \mathrm{Var}_{s^2}[\mathcal{M}(V)] &= \frac{\omega_{\mathcal{M}} \omega_T}{\omega_{\mathcal{M}} + \omega_T} \end{aligned}$$

8.78.5 Choice of prior values

For the first- and second-order quantities μ, Σ and Γ in the case of the population mean update, and ω_Y in the case of the population variance update we suggest relying on standard belief specification or elicitation techniques. These quantities are simply means, variances and covariances of observable quantities so obtaining appropriate prior values via direct specification or numerical investigation of related problems both provide relevant methods of assessment.

The quantities which are most challenging to specify beliefs about are the fourth-order quantities $\omega_{\mathcal{M}} =$ Var$[\mathcal{M}(V)]$, and $\omega_{\mathcal{R}} =$ Var$[\mathcal{R}(V)]$ required by the population variance update. Direct assessment or specification of these quantities is challenging, so we briefly describe an heuristic method for making such belief assessments.

Comparison to known distributions

Let us first consider $\omega_{\mathcal{R}} =$ Var$[\mathcal{R}(V)]$. This quantity represents our judgements about the shape of the distribution of the Y_i. One method of assessment is therefore to relate $\omega_{\mathcal{R}}$ to the **kurtosis** of that distribution, and then specify an appropriate value by comparison to the shape of other known distributions.

Suppose that we consider that the population variance acts as a scale parameter for the residuals $\mathcal{R}_i(Y)$ such that $\mathcal{R}_i(Y) = \sqrt{\mathcal{M}(V)}Z_i$, where the Z_i are independent standardised quantities with zero mean and unit variance which are independent of the value of $\mathcal{M}(V)$. Then $\mathcal{R}_i(V) = \mathcal{M}(V)(Z_i^2 - 1)$, and so $\omega_{\mathcal{R}} =$ Var$[\mathcal{R}_i(V)] = (\omega_{\mathcal{M}} + \omega_Y^2)Var[Z_i^2] = (\omega_{\mathcal{M}} + \omega_Y^2)(\kappa - 1)$, where $\kappa =$ Kur(Z_i) is the kurtosis of Z_i. If we believe that the Z_i were Gaussian in distribution, this would suggest a value of $\kappa = 3$. Similarly, a Uniform distribution suggests $\kappa = 1.8$ and a t-distribution with unit variance and ν degrees of freedom gives $\kappa = 3(\nu - 1)/(\nu - 4)$. In general, higher values for

κ (and hence $\omega_{\mathcal{R}}$), increase the proportion of variance in $\mathcal{M}(V)$ which cannot be resolved by observing data and so diminish the weight of the observations in update formula.

Proportion of variance resolved

Given a choice for $\omega_{\mathcal{R}}$, we now must determine an appropriate value for $\omega_{\mathcal{M}}$ to complete our belief specification. We briefly discuss two possible methods, the first arising from considerations of the effectiveness of the update. Let us write $\omega_{\mathcal{M}} = c\omega_Y^2$ for some $c > 0$, then the problem reduces to selection of a value of c. We can try to assess c by considering the proportion of uncertainty remaining in the population variance after the update, which is given by

$$\frac{\text{Var}_{s^2}[\mathcal{M}(V)]}{\text{Var}[\mathcal{M}(V)]} = \frac{1}{1 + \frac{n-1}{\phi}\frac{c}{c+1}}$$

which decreases monotonically as a function of c, and where we define ϕ as

$$\phi = \frac{1}{n}\{(n-1)\text{Var}[Z_i^2] + 2\}$$

which is fixed given n and κ. We can then explore our attitudes to the implications of differing sample sizes; for example, small values of c would suggest that any sample information will rapidly reduce our remaining variance as a proportion of the prior.

Equivalent sample size

An alternative approach for specifying $\omega_{\mathcal{M}}$ is to make a direct judgement on the worth of the prior information via the notion of equivalent sample size. We can express the adjusted expectation of the population variance as

$$\text{E}_{s^2}[\mathcal{M}(V)] = \alpha s^2 + (1-\alpha)\text{E}[\mathcal{M}(V)],$$

with

$$\alpha = \frac{\omega_{\mathcal{M}}}{\omega_{\mathcal{M}} + \omega_T}.$$

Suppose that we consider that the prior information we have about the population variance is worth a notional sample size of m, and that we collect observations of a sample of size n. In which case, it would be reasonable to adjust our beliefs with a weighting given by

$$\alpha = \frac{n}{n+m}.$$

By examining our beliefs about the relative merits of the prior and sample information we can make an assessment for an appropriate value of $\omega_{\mathcal{M}}$. In fact, this method of specification is equivalent to the alternative method discussed above.

8.78.6 References

- Goldstein, M. and Wooff, D. A. (2007), Bayes Linear Statistics: Theory and Methods, Wiley.

8.79 Discussion: Theoretical aspects of Bayes linear

8.79.1 Overview

The *Bayes linear* approach is similar in spirit to conventional *Bayes* analysis, but derives from a simpler system for prior specification and analysis, and so offers a practical methodology for analysing partially specified beliefs for larger problems. The approach uses expectation rather than probability as the primitive for quantifying uncertainty; see De Finetti (1974, 1975).

For a discussion on the differences between Bayes linear and full Bayes approaches to emulation see *AltGPorBLEmulator*.

8.79.2 Notation

Given the vector X of random quantities, then we write $\mathrm{E}[X]$ as the expectation vector for X, and $\mathrm{Var}[X]$ as the variance-covariance matrix for the elements of X. Given observations D, we modify our prior expectations and variances to obtain *adjusted* expectations and variances for X indicated by a subscript, giving $\mathrm{E}_D[X]$, and $\mathrm{Var}_D[X]$.

Aspects of the relationship between the adjusted expectation $\mathrm{E}_D[X]$ and the conditional expectation $E[X|D]$ are discussed later in this page.

8.79.3 Foundations of Bayes linear methods

Let $C = (B, D)$ be a vector of random quantities of interest. In the Bayes linear approach, we make direct prior specifications for that collection of means, variances and covariances which we are both willing and able to assess. Namely, we specify $\mathrm{E}[C_i]$, $\mathrm{Var}[C_i]$, $\mathrm{Cov}[C_i, C_j]$ for all elements C_i, C_j in the vector C, $i \neq j$. Suppose we observe the values of the subset D of C. Then, following the Bayesian paradigm, we modify our beliefs about the quantities B given the observed values of D

Following Bayes linear methods, our modified beliefs are expressed by the *adjusted* expectations, variances and covariance for B given D. The adjusted expectation for element B_i given D, written $\mathrm{E}_D[B_i]$, is the linear combination $a_0 + \mathbf{a}^T D$ minimising $\mathrm{E}[B_i - a_0 - \mathbf{a}^T D)^2]$ over choices of $\{a_0, \mathbf{a}\}$. The adjusted expectation vector is evaluated as

$$\mathrm{E}_D[B] = \mathrm{E}[B] + \mathrm{Cov}[B, D]\mathrm{Var}[D]^{-1}(D - \mathrm{E}[D])$$

If the variance matrix $\mathrm{Var}[D]$ is not invertible, then we use an appropriate generalised inverse.

Similarly, the *adjusted variance matrix* for B given D is

$$\mathrm{Var}_D[B] = \mathrm{Var}[B] - \mathrm{Cov}[B, D]\mathrm{Var}[D]^{-1}\mathrm{Cov}[D, B]$$

Stone (1963), and Hartigan (1969) are among the first to discuss the role of such assessments in partial Bayes analysis. A detailed account of Bayes linear methodology is given in Goldstein and Wooff (2007), emphasising the interpretive and diagnostic cycle of subjectivist belief analysis. The basic approach to statistical modelling within this formalism is through second-order exchangeability.

8.79.4 Interpretations of adjusted expectation and variance

Viewing this approach from a full Bayesian perspective, the adjusted expectation offers a tractable approximation to the conditional expectation, and the adjusted variance provides a strict upper bound for the expected posterior variance, over all possible prior specifications which are consistent with the given second-order moment structure. In certain special cases these approximations are exact, in particular if the joint probability distribution of B, D is multivariate normal then the adjusted and conditional expectations and variances are identical.

In the special case where the vector D is comprised of indicator functions for the elements of a partition, ie each D_i takes value one or zero and precisely one element D_i will equal one, then the adjusted expectation is numerically equivalent to conditional expectation. Consequently, adjusted expectation can be viewed as a generalisation of de Finetti's approach to conditional expectation based on 'called-off' quadratic penalties, where we now lift the restriction that we may only condition on the indicator functions for a partition.

Geometrically, we may view each individual random quantity as a vector, and construct the natural inner product space based on covariance. In this construction, the adjusted expectation of a random quantity Y, by a further collection of random quantities D, is the orthogonal projection of Y into the linear subspace spanned by the elements of D and the adjusted variance is the squared distance between Y and that subspace. This formalism extends naturally to handle infinite collections of expectation statements, for example those associated with a standard Bayesian analysis.

A more fundamental interpretation of the Bayes linear approach derives from the temporal sure preference principle, which says, informally, that if it is necessary that you will prefer a certain small random penalty A to C at some given future time, then you should not now have a strict preference for penalty C over A. A consequence of this principle is that you must judge now that your actual posterior expectation, $\mathrm{E}_T[B]$, at time T when you have observed D, satisfies the relation $\mathrm{E}_T[B] = \mathrm{E}_D[B] + R$, where R has, a priori, zero expectation and is uncorrelated with D. If D represents a partition, then $E_D[B]$ is equal to the conditional expectation given D, and R has conditional expectation zero for each member of the partition. In this view, the correspondence between actual belief revisions and formal analysis based on partial prior specifications is entirely derived through stochastic relationships of this type.

8.79.5 References

- De Finetti, B. (1974), Theory of Probability, vol. 1, Wiley.
- De Finetti, B. (1975), Theory of Probability, vol. 2, Wiley.
- Goldstein, M. and Wooff, D. A. (2007), Bayes Linear Statistics: Theory and Methods, Wiley.
- Hartigan, J. A. (1969), "Linear Bayes methods," *Journal of the Royal Statistical Society*, Series B, 31, 446–454.
- Stone, M. (1963), "Robustness of non-ideal decision procedures," *Journal of the American Statistical Association*, 58, 480–486.

8.80 Discussion: The Best Input Approach.

8.80.1 Description and Background

As introduced in the variant thread on linking models to reality (*ThreadVariantModelDiscrepancy*), the most commonly used method of linking a model f to reality y is known as the *Best Input* approach. This page discusses the Best Input assumptions, highlights the strengths and weaknesses of such an approach and gives links to closely related discussions regarding the *model discrepancy* d itself, and to methods that go beyond the Best Input approach. Here we use the term model synonymously with the term *simulator*.

Notation

The following notation and terminology is introduced in *ThreadVariantModelDiscrepancy*. In accordance with standard *toolkit notation*, in this page we use the following definitions:

- x - inputs to the model
- $f(x)$ - the model function
- y - the real system value
- z - an observation of reality y
- x^+ - the 'best input' (see below)
- d - the model discrepancy (see below)

The full definitions of x^+ and d are given in the discussion below.

8.80.2 Discussion

A model is an imperfect representation of reality, and hence there will always be a difference between model output $f(x)$ and system value y. Refer to *ThreadVariantModelDiscrepancy* where this idea is introduced, and to the discussion page on model discrepancy (*DiscWhyModelDiscrepancy*) for the importance of including this feature in our analysis.

It is of interest to try to represent this difference in the simplest possible manner, and this is achieved by a method known as the Best Input approach. This approach involves the notion that were we to know the actual value, x^+, of the system properties, then the evaluation $f(x^+)$ would contain all of the information in the model about system performance. In other words, we only allow x^+ to vary because we do not know the appropriate value at which to fix the input. This does not mean that we would expect perfect agreement between $f(x^+)$ and y. Although the model could be highly sophisticated, it will still offer a necessarily simplified account of the physical system and will most likely approximate the numerical solutions to the governing equations (see *DiscWhyModelDiscrepancy* for more details). The simplest way to view the difference between $f^+ = f(x^+)$ and y is to express this as:

$$y = f^+ + d.$$

Note that as we are uncertain as to the values of y, f^+ and d, they are all taken to be random quantities. We consider d to be independent of x^+ and uncorrelated with f and f^+ (in the Bayes Linear Case) or independent of f (in the fully Bayesian Case). The Model Discrepancy d is the subject, in various forms, of *ThreadVariantModelDiscrepancy*. The definition of the model discrepancy given by the Best Input approach is simple and intuitive, and is widely used in computer modelling studies. It treats the best input x^+ in an analogous fashion to the parameters of a statistical model and (in certain situations) is in accord with the view that x^+ is an expression of the true but unknown properties of the physical system.

In the case where each of the elements of x correspond to a well defined physical property of the real system, x^+ can simply be thought of as representing the actual values of these physical properties. For example, if the univariate input x represented the acceleration due to gravity at the Earths surface, x^+ would represent the real system value of $g = 9.81ms^{-2}$.

It is often the case that certain inputs do not have a direct physical counterpart, but are, instead, some kind of 'tuning parameter' inserted into the model to describe approximately some physical process that is either too complex to model accurately, or is not well understood. In this case x^+ can be thought of as the values of these tuning parameters that gives the best performance of the model function $f(x)$ in some appropriately defined sense (for example, the best agreement between f^+ and the observed data). See the discussion pages on reification (*DiscReification*) and its theory (*DiscReificationTheory*) for further details about tuning parameters.

In general, inputs can be of many different types: examples include physical parameters, tuning parameters, aggregates of physical quantities, control (or variable) inputs, or decision parameters. The meaning of x^+ can be different for each type: for a decision parameter or a control variable there might not even be a clearly defined x^+, as we might want to simultaneously optimise the behaviour of f for all possible values of the decision parameter or the control variable.

The statement that the model discrepancy d is probabilistically independent of both f and x^+ (or uncorrelated with f or x^+ in the Bayes Linear case) is a simple and in many cases largely reasonable assumption, that helps ensure the tractability of subsequent calculations. It involves the idea that the modeller has made all the improvements to the model that s/he can think of, and that beliefs about the remaining inaccuracies of the model would not be altered by knowledge of the function f or the best input x^+; see further discussions in *DiscReification* and *DiscReificationTheory*.

8.80.3 Additional Comments

Although useful for many applications, the Best Input approach does break down in certain situations. If we have access to two models, the second a more advanced version of the first, then we cannot use the Best Input assumptions for both models. In this case x^+ would be different for each model, and it would be unrealistic to simultaneously impose the independence assumption on both models. An approach which resolves this issue by modelling relationships across models, known as Reification, is described in *DiscReification* with a more theoretical treatment given in *DiscReificationTheory*.

8.81 Discussion: Computational issues in building a Gaussian process emulator for the core problem

8.81.1 Description and Background

The procedure for building a *Gaussian process emulator* for the *core problem* is described in page *ProcBuildCoreGP*. However, computational problems can arise in implementing that procedure, and these problems are discussed here.

8.81.2 Inversion of the correlation matrix of the training sample

Problems arise primarily in the computation of the inverse, $A^{-1} \equiv c(D,D)^{-1}$, of the correlation matrix of the *training sample* data. This can happen with any form of correlation function when the training sample is large or when the *correlation function* exhibits a high degree of *smoothness* (e.g. large values of correlation length parameters) relative to the size of the design region. It is particularly problematic for highly *regular* correlation functions (such as the Gaussian form).

Non invertibility of the correlation matrix can imply that two or more points are so close in the input space, that offer little or no information to the emulator. A way of sidestepping this problem is to identify these points using the *pivoted Cholesky* decomposition. The result of this decomposition are two matrices, R, which is an upper triangular containing the Cholesky coefficients, and P, which is a permutation matrix. For example if

$$A = \begin{bmatrix} 1 & 0.1 & 0.2 \\ 0.1 & 3 & 0.3 \\ 0.2 & 0.3 & 2 \end{bmatrix}$$

then

$$R = \begin{bmatrix} 1.73 & 0.17 & 0.06 \\ 0 & 1.40 & 0.14 \\ 0 & 0 & 0.99 \end{bmatrix}$$

and

$$P = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \text{ or } piv = [2,3,1]$$

Of interest here is the permutation matrix P. The row number of the ‘1’ in the first column, shows the position of the element with the larger variance (in this example it is the second element). Similarly, the row number of the ‘1’ in the second column shows the position of the element with the largest variance, conditioned on the first element, and so forth. If the matrix A is non invertible, and therefore, non positive definite, the k last elements in the main diagonal of R will be zero, or very small. If this is the case, the design points that correspond to the row number where the ones appear in the last k columns of matrix P can be excluded from the design matrix, and the emulator can be built using the remaining points, without losing information in principle.

8.81.3 Matrix inversion using the Cholesky decomposition

The parameter estimation and prediction formulae presented in *ProcBuildCoreGP* contain a fair amount of matrix inversions. Although these can be calculated with general purpose inversion algorithms, there are more efficient ways of carrying out the inversion, by taking into account the special structure of these matrices; that is, by taking into account that the matrices or matrix expressions that need to be inverted are positive definite.

The majority of the matrix inverses come in two forms: a left inverse matrix multiplication (i.e. $A^{-1}f(D)$) and a quadratic term of the form $H^{\mathrm{T}}A^{-1}H$. Both of these forms can be efficiently calculated using the Cholesky decomposition and the left matrix division. We consider the Cholesky decomposition of a matrix to the product of a lower triangular matrix L and its transpose, i.e.

$$A = LL^{\mathrm{T}}$$

The left matrix division, which we denote with backslash (\), represents the solution to a linear system of equations; that is if $Ax = y$, then $x = A\backslash y$. Furthermore, the fact that the Cholesky decomposition results in triangular matrices, means that we can calculate expressions of the form $L\backslash y$ using backsubstitution, taking advantage of its efficiency.

Using the left matrix division and the Cholesky decomposition, expressions of the form $A^{-1}f(D)$ can be calculated as

$$A^{-1}f(D) \equiv L^{\mathrm{T}}\backslash(L\backslash f(D))$$

On the other hand, quadratic expressions of the form $H^{\mathrm{T}}A^{-1}H$ can be calculated using an intermediate vector w, as

$$w = L\backslash H, \qquad H^{\mathrm{T}}A^{-1}H \equiv w^{\mathrm{T}}w$$

The Cholesky decomposition is also useful in the calculation of logarithms of derivatives, yielding more numerically stable results. The logarithm of the derivative of A, which appears in various likelihood expressions, is best calculated using the identity

$$\ln|A| \equiv 2\sum \ln(L_{ii})$$

with L_{ii} being the i-th element on the main diagonal of L.

Example

As an example, we show how the expression for $\hat{\beta}$ can be calculated, using the Cholesky decomposition. The original expression is

$$\hat{\beta} = (H^{\mathrm{T}}A^{-1}H)^{-1}H^{\mathrm{T}}A^{-1}f(D)$$

This can be calculated with the following five steps

- $L = chol(A)$
- $w = L\backslash H$
- $Q = w^{\mathrm{T}}w \quad (= H^{\mathrm{T}}A^{-1}H)$
- $K = chol(Q)$
- $\hat{\beta} = K^{\mathrm{T}}\backslash(K\backslash H^{\mathrm{T}})(L^{\mathrm{T}}\backslash(L\backslash f(D)))$

Even though this implementation might be more cumbersome, it is numerically more stable compared to an implementation that uses general purpose matrix inversion routines.

8.81.4 High input dimensionality

Problems may also arise through having a large number of parameters to estimate. When the number of *simulator* inputs is large, there are many elements in the correlation function's *hyperparameter* vector δ, and it can be computationally very demanding then to find a suitable single estimate or to compute a sample using Markov chain Monte

Carlo. Experience suggests that in practice in a complex simulator with many inputs many of those inputs will be more or less redundant, having negligible influence on the output in the practical context of interest. Thus it is important to be able to apply a preliminary *screening* process to reduce the input dimension - see the screening topic thread (*ThreadTopicScreening*).

8.81.5 Additional Comments

Techniques for addressing these problems are being developed within *MUCM*, and will be incorporated in this page in due course.

8.82 Discussion: The core problem

8.82.1 Description and Background

The *MUCM* toolkit is *structured* in various ways, one of which is via a number of threads that take the user through building and using various kinds of *emulator*. The core thread concerns the simplest kind of emulator, while variant threads build on the methods in the core thread to address more complex problems.

The situation to which the core thread applies is called the core problem, and is characterised as follows:

- We are only concerned with one *simulator*.
- The simulator only produces one output, or (more realistically) we are only interested in one output.
- The output is *deterministic*.
- We do not have observations of the real world process against which to compare the simulator.
- We do not wish to make statements about the real world process.
- We cannot directly observe derivatives of the simulator.

This discussion page amplifies on this brief specification, and indicates how each of these requirements might be relaxed in the variant threads.

8.82.2 Discussion

We consider each of the core problem requirements in turn.

One simulator

Practical problems concerning the use of *simulators* may involve more than one simulator, in which case we need to build several connected emulators.

An example of one such situation which often arises is if we wish to use climate simulation to tell us about the consequences of future climate change, for then we have a number of climate simulators available. We may wish to use several such simulators in order to contrast their predictions, or to combine them to make some kind of average prediction.

Another situation in which multiple simulators are of interest is when a single simulator may be implemented using different levels of detail. For instance, a simulator of an oilfield will represent the geology of the field in terms of blocks of rock, each block having its own properties (such as permeability). If it is run using a small number of very large blocks, the simulator will run quickly but will be a poor representation of the underlying real-world oilfield. It can also be run with an enormous number of very small blocks, which will represent the real oilfield much more accurately but the computer program will take many days to run. It is then often of interest to combine a few runs of the accurate simulator with a larger number of runs of a coarser simulator. A version of this problem is addressed in the variant thread for two level emulation of the core model using a fast approximation (*ThreadVariantTwoLevelEmulation*).

In the core problem, we suppose that we are only interested in one simulator. In a future release of the toolkit we intend to introduce a variant thread ThreadVariantMultipleSimulators describing how to handle the emulation of multiple simulators.

One output

Simulators almost invariably produce many outputs. For example, a climate simulator may output various climate properties, such as temperature and rainfall, at each point in a grid covering the geographical region of interest and at many points in time. We will often wish to address questions that relate to more than one output.

However, for the core problem we suppose that interest focuses on a single output, so that we only wish to emulate that one output. The variant thread for the analysis of a simulator with multiple outputs using Gaussian process methods (*ThreadVariantMultipleOutputs*) presents ways to handle the emulation of multiple outputs. A special case of a dynamic simulator producing a time series of outputs is considered in the variant thread for dynamic emulation (*ThreadVariantDynamic*).

Note that the restriction to a single output is weakened by the fact that we can be flexible about what we define a simulator output to be. For instance, a simulator may produce temperature forecasts for each day in a year, whereas we are interested in the mean temperature for that year. Whilst the mean temperature is not actually produced as an output by the simulator we can readily compute it from the actual outputs. We can formally treat the combination of the simulator with the post-processing operation of averaging the temperatures as itself a simulator that outputs mean temperature. In general, the output of interest may be any function of the simulator's actual outputs.

Deterministic

Simulators can be *deterministic* or *stochastic*. A deterministic emulator returns the same output when we run it again at the same inputs, and so the output may be regarded as a well-defined function of the inputs.

The core problem supposes that the output which we wish to emulate is deterministic. In a future release of the toolkit we intend to introduce a variant thread ThreadVariantStochastic concerned with extending the core methods to stochastic simulators.

No real-world observations

An important activity for many users of simulators is to compare the simulator outputs with observations of the real-world process being simulated. In particular, we often wish to use such observations for *calibration* of the simulator, i.e. to learn which values of certain inputs are the best ones to use in the simulator.

The core problem assumes that we do not have real-world observations that we wish to apply to test or calibrate the simulator. If we do have observations of the real world to consider then we must build a representation of the relationship between the simulator and the real system - e.g. how good is the simulator? This can be handled by the same techniques as are used for multiple simulators, by thinking of the real system as a perfect simulator. Hence the appropriate thread for this case will be ThreadVariantMultipleSimulators.

No real-world statements

A related assumption in the core problem is that we do not wish to make statements about the real-world system. Of course, we very often *do* want to make such statements, and indeed the whole purpose of using simulators is naturally seen as to help us to predict, understand or control the real system. However, to make such statements we must again build a representation of how the simulator relates to reality, and so this will again be handled in ThreadVariantMultipleSimulators.

Although the core problem allows us only to make statements about the simulator, not reality, we can predict what outputs the simulator would produce at different input configurations and we can analyse in various ways how the simulator responds to uncertain inputs. These are important tasks in their own right.

No observations of derivatives

We will build the emulator using a training set of runs of the simulator, so that we know the output values that are obtained with the input configurations in the training data. Some simulators are able to deliver not just the output in question but also the (partial) derivatives of that output with respect to one or more (or all) of the inputs. This is often done by the creation of what is called an adjoint to the simulator.

The core problem assumes that we do not have such derivatives available. Obviously, if we had some derivatives in addition to the other training data, we should be able to create a more accurate emulator. In a future release of the toolkit this is expected to be dealt with in a variant thread ThreadVariantDerivatives.

8.82.3 Additional Comments

The ways that the toolkit is structured, and in particular the threads, are discussed in the *Toolkit structure* page.

8.83 Discussion: Technical issues in training sample design for the core problem

8.83.1 Description and Background

A number of alternatives for the design of a training sample for the *core problem* are presented in the alternatives page on training sample design for the core problem (*AltCoreDesign*), but the emphasis there is on presenting the simplest and most widely used approaches with a minimum of technical detail. This discussion page goes into more detail about the alternatives.

In particular,

- *AltCoreDesign* assumes that the region of interest has been transformed to $[0, 1]^p$ by simple linear transformations of each *simulator* input. We discuss here the formulation of the region of interest and various issues around transforming it to a unit hypercube in this way.
- *AltCoreDesign* presents a number of ways to generate designs that have a space-filling character. We discuss here the principles of optimal design, various simplified criteria for generating training samples, and how these are related to the space-filling designs.

8.83.2 Discussion

The design region

The design region, which we denote by $\mathcal{X}$, is a subset of the space of possible input configurations that we will confine our training design points $x_1, x_2, \ldots, x_n$ to lie in.

Specifying the region

The design region should cover the region over which it is desired to *emulate* the simulator. This will usually be a relatively small part of the whole space of possible input configurations. Simulators are generally built to represent a wide range of practical contexts, whereas the user is typically interested in just one such context. For example, a simulator that simulates the hydrology of rainwater falling in a river catchment will have inputs whose values can be defined to represent a wide range of catchments, whereas the user is interested in just one catchment. The range of input parameter values of interest would then be just the values that might be appropriate to that catchment. Because of uncertainty about the true values of the inputs to describe that catchment, we will wish to build the emulator over a range of plausible values for each uncertain input, but this will still be a small part of the range of values that the simulator can accept.

$\mathcal{X}$ is usually specified by thinking of a range of plausible values for each input and then defining $\mathcal{X} = \mathcal{X}_1 \times \mathcal{X}_2 \times \ldots \times \mathcal{X}_p$, where $\mathcal{X}_i$ is a finite interval of plausible values for input i. We then say that $\mathcal{X}$ is rectangular. In practice, we may not be able to limit the plausible values for a given input, in the sense that in theory that input might conceivably take any value, but it would be extremely surprising if it lay outside a well-defined and restricted region. Then it would be inefficient to try to emulate the simulator over the whole range of theoretical possible values of this input, and we would define its $\mathcal{X}_i$ to cover only the range that the input is likely to lie in. This is why we have used the phrase “range of plausible values.”

It may be that some inputs are known or specified precisely, so that some $\mathcal{X}_i$ could contain just a single value. However, we can effectively ignore these inputs and act as if the simulator is defined with these parameters fixed. Similarly, we can ignore any inputs whose values must for this context equal known functions of other inputs. Throughout the *MUCM* toolkit, the number of inputs (denoted by p) is always the number of inputs that can take a range of values (even if all other inputs were to be fixed). So $\mathcal{X}$ will always be of dimension p.

However, $\mathcal{X}$ does not have to be rectangular. A simulator of plant growth might include inputs describing the soil in terms of the percentages of clay and sand. Not only must each percentage be between 0 and 100, but the sum of the two percentages must be less than or equal to 100. This latter constraint imposes a non-rectangular region for these two inputs. *AltCoreDesign* does not discuss design over non-rectangular regions. Some possible generic approaches are:

- Generate a design over a larger rectangular region containing $\mathcal{X}$ and reject any points lying outside $\mathcal{X}$. This would seem a sensible approach if $\mathcal{X}$ is “almost” rectangular, so that not many points will be rejected.
- Develop specific design methods for non-rectangular regions. This is a topic that is being studied in MUCM, but such methods are not currently available in the toolkit.
- Transform $\mathcal{X}$ to a rectangular region. In the example of the percentages of clay and sand in soil, two kinds of transformations come to mind. If we denote these two soil inputs by s_1 and s_2, then the first transformation is to define the inputs to be s_1 and $100\, s_2/(100 - s_1)$. In terms of these two inputs, $\mathcal{X} = [0, 100]^2$. Non-linear one-to-one transformations like this are discussed below.

The second transformation option for the soil inputs example is to generate a design over the unrestricted range of [0,100] for both s_1 and s_2 and to reflect any point having $s_1 + s_2 > 100$ by replacing s_1 by $100 - s_1$ and s_2 by $100 - s_2$. This works because in this case there is a simple one-to-two mapping between $\mathcal{X}$ and the rectangular region, but although this might be indicative of tricks that could be applied in other situations it is not a generic approach to the problem of non-rectangular regions.

Transforming to a unit hypercube

AltCoreDesign assumes that $\mathcal{X}$ is rectangular and that each $\mathcal{X}_i$ is transformed to $[0, 1]$ by a simple linear transformation. More generally, we could use nonlinear transformations of individual inputs or, as we have seen, transformations of two or more inputs together. Any one-to-one transformation which turns $\mathcal{X}$ into $[0, 1]^p$ could be used in conjunction with the design procedures of *AltCoreDesign* to produce a training sample design. However, the transformation that is used has implications for the optimality criteria considered below, and in particular for the appropriateness of simple space-filling designs.

The transformation needs to be one-to-one so that we can transform each design point back into the original input space in order to run the simulator at those inputs. We denote the back transformation by t, so that the design point x_j converts to the point $t(x_j)$ in the space of the simulator inputs.

Optimality criteria

In principle, what makes a good design depends on what we know about the simulator to begin with. Hence the modelling decisions that we take with regard to the mean and covariance functions, plus what we also express in terms of prior information about the *hyperparameters* in those functions - see the core threads: the thread for the analysis of the core model using Gaussian process methods (*ThreadCoreGP*) and the thread for the Bayes lineaar emulation for the core model (*ThreadCoreBL*).

The formal way to develop an optimal design is using Bayesian decision theory. This ideal is not feasible for the problem of choosing a training sample for building an emulator. Nevertheless, we can bear in mind the broad nature of such a solution when considering simplified criteria.

Principles of optimal design for a training sample

A good design will enable us to build a good emulator, one that predicts the simulator's output accurately. The aim of the design is to reduce uncertainty about the simulator. Our uncertainty about the simulator before obtaining the training sample data has two main components.

- We are uncertain about the values of the hyperparameters - β (the hyperparameters of the mean function), σ^2 (the variance parameter in the covariance function) and δ (the other hyperparameters in the covariance function). This uncertainty is expressed in their prior distributions or moments.
- We would still be uncertain about the precise output that the simulator will produce for any given input configuration, even if we knew the hyperparameters. This uncertainty is expressed in the Gaussian process (in the fully *Bayesian* approach of *ThreadCoreGP*), or in the equivalent *Bayes linear* second-order moments interpretation of the mean and covariance functions (in the approach of *ThreadCoreBL*).

What features of the design will help us to learn about these things? If we first consider the second kind of uncertainty, regarding the shape of the simulator output as a function of its inputs, we should have design points that are not too close together. This is because points very close together are highly correlated, so that one could be predicted well from the other. Having points too close together implies some redundancy in being able to predict the function.

Good design to learn about the parameters β of the mean function depends on the form of the function, which is discussed in the alternatives function on the emulator prior mean function (*AltMeanFunction*). If we have a constant mean, $m(x) = \beta$, the location of the design points is irrelevant and all designs are equally good. If we have a mean function of the form $m(x) = \beta_1 + \beta_2^T x$ expressing a linear trend in each input, a good design will concentrate points in the corners of the design space, to learn best about the slope parameters β_2. If the mean function includes terms that are quadratic in the inputs, we will concentrate design points on more of the corners and also in the centre of the space.

To learn about σ^2, again it is of little importance where the design points are placed. To learn about δ, however, we will generally need pairs of points that are at different distances apart, from being very close together to being far apart.

Basing design criteria on predictive variances

The primary objective of the design is that after we have run the model at the training sample design points and fitted the emulator it will predict the simulator output at other input points accurately. This accuracy is determined by the posterior predictive variance at that input configuration. In the procedure page for building a Gaussian process emulator for the core problem (*ProcBuildCoreGP*) we find two formulae for the predictive variance conditional on hyperparameters. The full Bayesian optimal design approach would require the *unconditional* variances. These would be very complicated to compute and would depend not only on the design but also on the simulator outputs that we observed at the design points. These values are of course unknown at the time of creating the design, and so the full Bayesian design process would need to average over the prior uncertainty about those observations. It is this that makes proper optimal design impractical for this problem. We will instead base the design criteria on the conditional variance formulae in *ProcBuildCoreGP*. It is important to recognise, however, that in doing so our design will not be chosen with a view to learning about the parameters that we have conditioned on.

In the general case, conditional on the full set of hyperparameters $\theta = \{\beta, \sigma^2, \delta\}$ we have the variance function $v^*(x,x) = \sigma^2 c^{(1)}(x)$, where we define

$$c^{(1)}(x) = c(x,x) - c(x)^T A^{-1} c(x).$$

When the mean function takes the linear form and we have weak prior information on β and σ^2, then conditional only on δ we have the variance function $v^*(x,x) = \widehat{\sigma}^2 c^{(2)}(x)$, where now we define

$$c^{(2)}(x) = c^{(1)}(x) + c(x)^T A^{-1} H \left(H^T A^{-1} H\right)^{-1} H^T A^{-1} c(x).$$

In both cases we have a constant multiplier, σ^2 or its estimate $\widehat{\sigma}^2$. As discussed above, the details of the design have little influence on how well σ^2 is estimated, so we consider as the primary factor for choosing a design either $c^{(1)}(.)$ or $c^{(2)}(.)$ as appropriate.

Notice that neither formulae involves the to-be-observed simulator outputs from the design. The matrix A and the function $c(.)$ are defined in *ProcBuildCoreGP* as depending only on the correlation function $c(.,.)$, while the matrix H depends only on the assumed structure of the mean function. The only hyperparameters that are required by either formula are the vector of correlation function hyperparameters δ. It is therefore possible to base design criteria on either $c^{(1)}(x)$ or $c^{(2)}(x)$ if we are prepared to specify a prior estimate for δ.

The difference between the two functions is that $c^{(1)}(x)$ arises from the predictive covariance conditioned on all the hyperparameters, and so basing a design criterion on this formula will ignore learning about β. In contrast, the second term in $c^{(2)}(x)$ expresses specifically the learning about β in the case of the linear mean function. Neither allows for learning about δ.

The effect of transformation

Before discussing specific design criteria based on these functions, we return to the assumption that the input space has been transformed to the unit hypercube. This is important because we are proposing to use formulae which depend on the form of the correlation function and, in the case of $c^{(2)}(x)$ also on the form of the mean function. Both of these functions are for the purposes of our design problem defined on the unit cube, not on the original input space. If the correlation function in the original input space is $c^0(.,.)$ then the correlation function in the unit cube design space has the form $c(x,x') = c^0(t(x), t(x'))$.

Now if the transformation is a simple linear one in each dimension, then all the correlation functions considered in the alternatives page on emulator prior correlation function (*AltCorrelationFunction*) would have the same form in the transformed space, with only the correlation lengths (as expressed in δ) changing. Similarly, if the mean function has the linear form then this form is also retained in the transformed space. This is why in *AltCoreDesign* it is assumed that such a transformation is all that is required to achieve the unit hypercube design space. It is then not necessary to discuss the fact that we have potentially different forms of correlation and mean function in the transformed space.

For more complex cases, the distinction cannot be ignored. In the case of the mean function, a belief that the simulator output would respond roughly quadratically to a certain input would not then hold if we made a nonlinear transformation of that input. Unless we can realistically assume that the expected relationship between the output and the inputs has the linear form $h(x)^T\beta$ in the *transformed design space,* we cannot use $c^{(2)}(.,.)$.

The correlation function will often be less sensitive to the transformation, in the sense that we may not find it easy to say whether a specific form (such as the Gaussian form, see *AltCorrelationFunction*) would apply in the original input space or in the transformed design space. Indeed, transformation may make the simple stationary correlation functions in *AltCorrelationFunction* more appropriate (see also the discussion page on the Gaussian assumption (*DiscGaussianAssumption*)).

Specific criteria

Having said that we wish to minimise the predictive variance, the question arises: for which value(s) of x? The usual answer is to minimise the predictive variance integrated over the whole of the design space. This gives us the two criteria

$$C_I^{(u)}(D) = \int_{[0,1]^p} c^{(u)}(x)dx,$$

for $u = 1, 2$. These are the integrated predictive variance criteria.

The integration in the above formula gives equal weight to each point in the design space. There may well be situations in which we are more interested in achieving high accuracy over some regions of the space than over others. Then we can define a weight function $\omega(.)$ and consider the more general criteria

$$C_W^{(u)}(D) = \int_{R^p} c^{(u)}(x)\omega(x)dx,$$

for $u = 1, 2$. Notice now that we integrate not just over the unit hypercube but over the whole of p-dimensional space. This recognises the fact that although we have constrained the design space to cover all of the genuinely plausible values of the inputs we may still have some interest in predicting outside that range. These are the weighted predictive variance criteria.

How should we expect designs created under the various criteria to differ? First we note that designs using $C_I^{(2)}(D)$ or $C_W^{(2)}(D)$ take account of the possibility of learning about β and so can be expected to yield more design points towards the edges of the design space than their counterparts using $C_I^{(1)}(D)$ or $C_W^{(1)}(D)$. Second, the weighted criteria should produce more points in areas of high $\omega(x)$. However, both of these effects are moderated by the fact that points very close together are wasteful, since they provide almost the same information.

Hence we may expect designs to differ appreciably under the various criteria when the design size n is small, so that extra points in an area need not be so close together as to be redundant. But for large designs, all of these criteria are likely to yield fairly evenly spread designs.

All of these criteria are relatively computationally demanding to implement in practice (for instance when using the optimised Latin hypercube design method, as described in the procedure page (*ProcOptimalLHC*)). Some theory using entropy arguments shows that minimising the criterion $C_I^{(1)}(D)$ is very similar to maximising the uncertainty in the design points, leading to the entropy criterion (also known as the D-optimality criterion), see *AltOptimalCriteria*.

$$C_E(D) = |A|,$$

which is much quicker to compute. Whereas low values of the other criteria are good, we aim for high values of $C_E(D)$.

All of these criteria can be used in the optimised Latin hypercube method, *ProcOptimalLHC*. This will usually produce designs that are close to optimal according to the chosen criterion, unless the optimal design is very far from evenly spread. In that case, searching for the best Latin hypercube is less likely to produce near-optimal designs, and other search criteria should be employed.

Prior choices of correlation hyperparameters

As discussed above, implementation of any of the above criteria requires a prior estimate of δ. In general this requires careful thought, but it is simplified if we have a Gaussian or exponential power form of correlation in the design space. For the Gaussian form we require only estimates of the correlation lengths in each input dimension, while for the exponential power form we also need estimates of the power hyperparameters in each dimension. For the latter, power parameters of 2 in each dimension reduce the exponential power form to the Gaussian, and would be appropriate if we expect smooth differentiability with respect to each input. (Note that to make this choice for the design stage does not imply an assumption of a Gaussian correlation function when the emulator is built.) Otherwise a value between 1 and 2, e.g. 1.5, would be preferred.

Correlation length parameters are all relative to the $[0, 1]$ range of values in each dimension. Typical values might be 0.5, suggesting a relatively smooth response to an input over that range. A lower value, e.g. 0.2, would be appropriate for an input that was thought to be strongly influential.

The choice of correlation length parameters in particular can influence the design. Assigning a lower correlation length to one input will tend to produce a design with shorter distances between points in this dimension.

Space-filling designs

Unless we specify unequal correlation lengths, or use a weighted criterion with a weight function that is very far from uniform over the design space, then we can expect all of the design criteria to produce designs with points that are more or less evenly spread over $[0, 1]^p$. This leads to a further simplification in design, by ignoring the formal predictive variance or entropy criteria above and simply choosing a design that has this even spread property. Such designs are called space-filling, and these are the design methods that are presented as the approaches in *AltCoreDesign*.

8.83.3 Additional Comments

Further background on designs, particularly on model based *optimal design*, can be found in the topic thread *ThreadTopicExperimentalDesign*.

This is an area of active research within MUCM. New findings and guidance may be added here in due course.

8.84 Discussion: Design of a validation sample

8.84.1 Description and Background

Having built an *emulator*, it is important to *validate* it by checking that the statistical predictions it makes about the *simulator* outputs agree with the actual simulator outputs. This validation process is described for the *core problem* in the procedure for validating a Gaussian process emulator (*ProcValidateCoreGP*), and involves comparing emulator predictions with data from a validation sample. We discuss here the principles of *designing* a validation study.

8.84.2 Discussion

A valid emulator should predict the simulator output with an appropriate degree of uncertainty. Validation testing checks whether, for instance, when the emulator's predictions are more precise (i.e. have a smaller predictive variance) the actual simulator output is closer to the predictive mean than when the emulator predictions are more uncertain. The predictions will be most precise when the emulator is asked to predict for an input point x' that is close to the training sample points that were used to build the emulator; they will be largest when predicting at a point x' that is far from all training sample points (relative to the estimated correlation lengths). Hence a good validation design will mix design points close to training sample points with points far apart. In particular, design points close to training sample points (or to each other) are highly sensitive to the emulator's estimated correlation function hyperparameters (such as correlation length parameters), which is an important aspect of validation.

A validation design comprises a set of n' design points $D' = \{x'_1, x'_2, \ldots, x'_{n'}\}$. There is little practical understanding yet about how big n' should be, or how best to achieve the required mix of design points, but some considerations are as follows.

When the simulator is slow to run, it is important to keep the size of the validation sample as small as possible. Nevertheless, we typically need enough validation points to validate against possibly poor estimation of the different hyperparameters, and we should expect to have at least as many points as there are hyperparameters. Whereas it is often suggested that the training sample size, n, should be about 10 times the number of inputs, p, we tentatively suggest that for validation we need $n' \geq 3p$.

To achieve a mix of points with high and low predictive variances, one approach is to generate a design in which the predictive variances of the *pivoted Cholesky* decomposition is maximised. This could be done by using this criterion in an *optimised Latin hypercube* design procedure.

Another idea is to randomly select a number of the original training design points, and place a validation design point close to each. For instance, if we selected a training sample point x then we could place the validation point x' randomly within the region where the correlation function $c(x, x')$ (with estimated values for δ) exceeds 0.8, say. The number of these points should be at least p. Then the remaining points could be chosen as a random *Latin hypercube*.

8.84.3 Additional Comments

This is a topic of ongoing research in *MUCM*, and we expect to add to this page in due course.

8.85 Discussion: The GP covariance function

8.85.1 Description and Background

In the fully *Bayesian* approach an *emulator* is created based on a *Gaussian process* (GP), and one of the principal steps in building the emulator is to specify the GP's covariance function. The covariance function also has an analogous role in the *Bayes linear* approach. See the thread for the analysis of the core model using Gaussian process methods (*ThreadCoreGP*) and the thread for Bayes linear emulation for the core model (*ThreadCoreBL*) for emulation of the *core problem* from the fully Bayesian and Bayes linear approaches, respectively.

8.85.2 Discussion

The covariance function gives the prior covariance between the simulator output at any given vector of input values and the output at any other given input vector. When these two vectors are the same, the covariance function specifies the prior variance of the output at that input vector, and so quantifies prior uncertainty about the simulator output. In principal, this function can take a very wide variety of forms.

Within the *MUCM* toolkit, we make an assumption of constant prior variance, so that in the case of a single output (as in the core problem) the covariance function has the form $\sigma^2 c(\cdot, \cdot)$, where $c(\cdot, \cdot)$ is a correlation function having the property that, for any input vector x, $c(x, x) = 1$. Hence the interpretation of σ^2 is as the variance of the output (at any input). The correlation function generally depends on a set of *hyperparameters* denoted by δ, while σ^2 is another hyperparameter.

When building a *multivariate GP* emulator for a simulator with multiple outputs, the covariance between the sets of outputs at inputs x and x' is a matrix which is assumed to have the *separable* form $\Sigma c(\cdot, \cdot)$, where now Σ is a between-outputs covariance matrix but $c(\cdot, \cdot)$ is a correlation function over the input space with the same interpretation for each output as in the single output case. See the variant thread for the analysis of a simulator with multiple outputs using Gaussian process methods (*ThreadVariantMultipleOutputs*).

8.85.3 Additional Comments

More generally, we might relax the assumption that the variance is constant across the input space. However, it is important for the resulting covariance function to be valid, in the sense that the implied variance for any linear combination of outputs at different input points remains positive.

One valid covariance function in which the variance is not constant takes the form $u(x)u(x')c(x, x')$, where $c(\cdot, \cdot)$ is a correlation function (see the alternatives page on emulator prior correlation function (*AltCorrelationFunction*)) as before, while $u(x)^2$ is the variance of the output at inputs x, and can now vary with x. In general, $u(\cdot)$ would be modelled using further hyperparameters.

8.86 Discussion: Decision-based Sensitivity Analysis

8.86.1 Description and background

The basic ideas of *sensitivity analysis* (SA) are presented in the topic thread on sensitivity analysis (*ThreadTopicSensitivityAnalysis*). We concentrate in the *MUCM* toolkit on probabilistic SA, but the reasons for this choice and alternative approaches are considered in the discussion page *DiscWhyProbabilisticSA*. In probabilistic SA, we assign a probability density function $\omega(x)$ to represent uncertainty about the inputs.

Decision-based sensitivity analysis is a form of probabilistic sensitivity analysis in which the primary concern is for how uncertainty in inputs impacts on a decision. We suppose that the output of a *simulator* is an input to this decision, and because of uncertainty about the inputs there is uncertainty about the output, and hence uncertainty as to which is the best decision. Uncertainty about an individual input is important if by removing that uncertainty we can expect to make much better decisions. We develop here the principal measures of influence and sensitivity for individual inputs and groups of inputs.

Notation

The following notation and terminology is introduced in *ThreadTopicSensitivityAnalysis*.

In accordance with the standard *toolkit notation*, we denote the simulator by f and its inputs by x. The focus of SA is the relationship between x and the simulator output(s) $f(x)$. Since SA also typically tries to isolate the influences of individual inputs, or groups of inputs, on the output(s), we let x_j be the j-th element of x and will refer to this as the j-th input, for $j = 1, 2, \ldots, p$, where as usual p is the number of inputs. If J is a subset of the indices $\{1, 2, \ldots, p\}$, then x_J will denote the corresponding subset of inputs. For instance, if $J = \{2, 6\}$ then $x_J = x_{\{2,6\}}$ comprises inputs 2 and 6, while x_j is the special case of x_J when $J = \{j\}$. Finally, x_{-j} will denote the whole of the inputs x *except* x_j, and similarly x_{-J} will be the set of all inputs except those in x_J.

Formally, $\omega(x)$ is a joint probability density function for all the inputs. The marginal density function for input x_j is denoted by $\omega_j(x_j)$, while for the group of inputs x_J the density function is $\omega_J(x_J)$. The conditional distribution for x_{-J} given x_J is $\omega_{-J|J}(x_{-J}\,|\,x_J)$. Note, however, that it is common for the distribution ω to be such that the various inputs are statistically independent. In this case the conditional distribution $\omega_{-J|J}$ does not depend on x_J and is identical to the marginal distribution ω_{-J}.

Note that by assigning probability distributions to the inputs we formally treat those inputs as random variables. Notationally, it is conventional in statistics to denote random variables by capital letters, and this distinction is useful also in probabilistic SA. Thus, the symbol X denotes the set of inputs when regarded as random (i.e. uncertain), while x continues to denote a particular set of input values. Similarly, X_J represents those random inputs with subscripts in the set J, while x_J denotes an actual value for those inputs.

8.86.2 Discussion

Decision under uncertainty

Decisions are hardest to make when their consequences are uncertain. The problem of decision-making in the face of uncertainty is addressed by statistical decision theory. Interest in *simulator* output uncertainty is often driven by the need to make decisions, where the simulator output $f(x)$ is a factor in that decision.

In addition to the joint probability density function $\omega(x)$ which represents uncertainty about the inputs, we need two more components for a formal decision analysis.

1. *Decision set*. The set of available decisions is denoted by $\mathcal{D}$. We will denote an individual decision in $\mathcal{D}$ by d.
2. *Loss function*. The loss function $L(d, x)$ expresses the consequences of taking decision d when the true inputs are x.

In order to understand the loss function, first note that we have shown it as a function of x, but it only depends on the inputs indirectly. The decision consequences actually depend on the output(s) $f(x)$, and it is uncertainty in the output that matters in the decision. But the output is directly produced by the input, and so we can write the loss function as a function of x. However, in order to evaluate $L(d, x)$ at any x we will need to run the simulator first to find $f(x)$.

The interpretation of the loss function is that it represents, on a suitable scale, a penalty for making a poor decision. To make the best decision we need to find the d that minimises the loss, but this depends on x. It is in this sense that uncertainty about (the simulator output and hence about) the inputs x makes the decision difficult. Uncertainty about x leads to uncertainty about the best decision. It is this decision uncertainty that is the focus of decision-based SA.

Note finally that for convenience of exposition we refer to a loss function and a single simulator output. In decision analysis the loss function is often replaced by a utility function, such that higher utility is preferred and the best decision is the one which maximises utility. However, we can just interpret utility as negative loss. Also, $f(x)$ can be a vector of outputs (all of which may influence the decision) - all of the development presented here follows through unchanged in this case.

Value of perfect information

We do not know x, so it is a random variable X, but we still have to take a decision. The optimal decision is the one that minimises the *expected* loss

$$\bar{L}(d) = \mathrm{E}[L(d, X)].$$

The use of expectation is important here, because it relates to our earlier statement that the loss function represents a penalty "on a suitable scale". The way that loss is defined must be such that expected loss is what matters. That is, a certain loss of 1 should be regarded as equivalent to an uncertain loss which is 0 or 2 on the flip of a coin. The expectation of the uncertain loss is $0.5 \times 0 + 0.5 \times 2 = 1$. The formulation of loss functions (or utility functions) is an important matter that is fundamental to decision theory - see references at the end of this page.

If we denote this optimal decision by M, then $\bar{L}(M) = \min_d \bar{L}(d)$.

Suppose we were able to discover the true value of x. We let $M_\Omega(x)$ be the best decision given the value of x. (In later sections we will use the notation $M_J(x_J)$ to denote the best decision given a subset of inputs x_J). For each value of x we have $L(M_\Omega(x), x) = \min_d L(d, x)$

The impact of uncertainty about X can be measured by the amount by which expected loss would be reduced if we could learn its true value. Given that X is actually unknown, we compare $\bar{L}(M)$ with the *expectation* of the uncertain $L(M_\Omega(X), X)$. The difference

$$V = \bar{L}(M) - \mathrm{E}[L(M_\Omega(X), X)]$$

is known as the expected value of perfect information (EVPI).

Value here is measured in the very real currency of expected loss saved. It is possible to show that V is always positive.

Value of imperfect information

In decision-based SA, we are interested particularly in the impact of uncertainty in individual inputs or a group of inputs. Accordingly, suppose that we were to learn the true value of input x_j. Then our optimal decision would be

$M_j(x_j)$, which minimises the expected loss conditional on x_j, i.e.

$$\bar{L}_j(d, x_j) = \mathrm{E}[L(d, X) \,|\, x_j].$$

As in the case of perfect information, we do not actually know the value of x_j, so the value of this information is the difference

$$V_j = \bar{L}(M) - \mathrm{E}[\bar{L}_j(M_j(X_j), X_j)].$$

More generally, if we were to learn the value of a group of inputs x_J, then the optimal decision would become $M_J(x_J)$, minimising

$$\bar{L}_J(d, x_J) = \mathrm{E}[L(d, X) \,|\, x_J],$$

and the value of this information is

$$V_J = \bar{L}(M) - \mathrm{E}[\bar{L}_J(M_J(X_J), X_j)].$$

In each case we have perfect information about x_j or x_J but no additional direct information about x_{-j} or x_{-J}. This is a kind of imperfect information that is sometimes called "partial perfect information". Naturally, the value of such information, V_j or V_J, will be less than the EVPI.

Another kind of imperfect information is when we can get some additional data S. For instance an input to the simulator might be the mean effect of some medical treatment, and we can get data S on a sample of patients. We can then calculate an expected value of sample information (EVSI); see references below.

8.86.3 Additional comments

Some examples to illustrate the concepts of decisions and decision-based SA are presented in the example page *ExamDecisionBasedSA*.

Relationship with variance-based SA

The basic decision-based SA measures are $M_J(x_J)$, which characterises the effect of a group x_J of inputs in terms of how the optimal decision changes when we fix x_J, and the value of information V_J, which quantifies the expected loss reduction from learning the value of x_J. It is no coincidence that the same symbols are used for the principal SA measures in variance-based SA, given in the discussion page *DiscVarianceBasedSA*. One of the examples in *ExamDecisionBasedSA* shows that variance-based SA can be obtained as a special case of decision-based SA. In that example, the decision-based measures $M_J(x_J)$ and V_J reduce to the measures with the same symbols in variance-based SA.

But although we can consider variance-based SA as arising from a special kind of decision problem, its measures are also very natural ways to think of sensitivity when there is no explicit decision problem. Thus $M_J(x_J)$ is the mean effect of varying x_J, while V_J can be interpreted both as the variance of $M_J(X_J)$ and as the expected reduction of overall uncertainty from learning about x_J. In decision-based SA, $M_J(x_J)$ and V_J are defined in ways that are most appropriate to the decision context but they do not have such intuitive interpretations. In particular, we note the following.

- V_J is not in general the variance of $M_J(X_J)$. Although that variance might be interesting in the more general decision context, the definition of V_J in terms of reduction in expected loss is more appropriate.
- In *DiscVarianceBasedSA*, the mean effect $M_J(x_J)$ can be expressed in terms of main effects and interactions, but these are not generally useful in the wider decision context. For instance, in some decision problems the set $\mathcal{D}$ of possible decisions is discrete, and it is then not even meaningful to take averages or differences.
- Even if $\omega(x)$ is such that inputs are statistically independent, there is no analogue in decision-based SA of the decomposition of overall uncertainty into main effect and interaction variances.

References

The following are some standard texts on statistical decision analysis, where more details about loss/utility functions, expected utility and value of information can be found.

Smith, J.Q. *Decision Analysis: A Bayesian Approach*. Chapman and Hall. 1988.

Clemen, R. *Making Hard Decisions: An Introduction to Decision Analysis*, 2nd edition. Belmont CA: Duxbury Press, 1996.

An example of decision based SA using emulators is given in the following reference.

Oakley, J. E. (2009). Decision-theoretic sensitivity analysis for complex computer models. *Technometrics* 51, 121-129.

8.87 Discussion: Exchangeable Computer Models

This page is under construction.

8.88 Discussion: Expert Assessment of Model Discrepancy

8.88.1 Description and Background

When linking a model to reality, an essential ingredient is the *model discrepancy* term d. The *assessment* of this term is rarely straightforward, but a variety of methods are available. This page describes the use of Expert Assessment. Here we use the term model synonymously with the term *simulator*.

8.88.2 Discussion

As is introduced in the variant thread on linking models to reality using model discrepancy (*ThreadVariantModelDiscrepancy*), the difference between the model $f(x)$ and the real system y is given by the model discrepancy term d, a precise definition of which is discussed in page *DiscBestInput*. d represents the deficiencies of the model: possible types are described in the model discrepancy discussion page (*DiscWhyModelDiscrepancy*).

We use the model discrepancy d to represent a difference about which we may have very little information. For example, if there are only a small number of observations with which to compare the model, then we may not have enough data to accurately assess d. The informal and formal methods for assessing d, discussed in pages *DiscInformalAssessMD* and *DiscFormalAssessMD*, may therefore not be appropriate. Of particular importance in this case is the method of Expert Assessment, whereby the subjective beliefs of the expert regarding the model discrepancy are represented by statements of uncertainty. These statements would usually be expressed in the form of statistical properties of d, or more likely, statistical properties of the distribution that d may be considered a realisation of.

While scientists are very familiar with the notion of uncertainty, they are generally unaccustomed to representing it probabilistically. This subjective approach (which underlies Subjective Bayesian Analysis) is extremely useful, especially in the case of model discrepancy when often very little data is available. The expert (the scientist) usually has reasonably detailed knowledge as to the deficiencies of the model, which the model discrepancy d is supposed to represent. All that is required is to convert this knowledge into statistical statements about d.

Assessment Procedure

A typical assessment might proceed along the following lines. Usually, one of the most important stages in the assessment of model discrepancy is to consider the defects in the model of which the scientist is already aware. As a first step, the scientist would be asked to list the main areas in which the model could be improved. If possible, improvements that are orthogonal/independent should be identified, that is improvements that could reasonably be considered independent of each other, perhaps due to them dealing with different physical parts of the model. Possible examples of improvements to the model might include: improved solution techniques, better treatment of aspects of the physics which are currently modelled, impact of aspects of physics currently ignored and inaccuracies in forcing functions. The more that each of these features can be broken down into manageable sub-units the better. This list of improvements can be based partly on upgrade plans for the model itself and partly on the scientific literature.

The scientist would then be asked to consider, for each such improvement, how much difference it would be likely to make to the computer output for a general input. It should be noted that the improved model sits between the current model and reality, as by definition, it must be considered to be closer to reality than the current model. Such considerations of improved models may lead to a fuller description of model discrepancy (which occurs in the Reification approach described in pages *DiscReification* and *DiscReificationTheory*), but for the purposes of this section they can be used to specify order of magnitude effects (for example standard deviations) for each of the contributions to the components of the model discrepancy d. If the scientist is unwilling to specify exact numbers for some of the required statistical quantities, an imprecise approach can be taken whereby only ranges are required for each quantity.

This general approach is useful for a single output, but, perhaps even more helpful for the multivariate case where we require a joint specification over many outputs. Judgements as to how each improvement to the model would affect subsets of the outputs would help determine the correlation structure over the outputs.

An important point to note is that the procedure of specifying model discrepancy is a serious scientific activity which requires careful documentation and a similar level of care to that for each other aspect of the modelling process.

Univariate Output

If we are dealing with a univariate output of the model (which is the case for the core model: see *ThreadCoreGP* and *ThreadCoreBL*), then we would follow the above procedure and would be required to assess each of the univariate contributions to the random quantity $d = y - f(x^+)$. In the fully probabilistic case, this involves *eliciting* the full distribution for each contribution. To achieve this, an appropriate choice of one of the standard distributions might be made (e.g the Normal Distribution), and the scientist would then be asked a series of questions designed to identify the parameters of the distribution (in this case the mean μ and standard deviation σ). Often these questions will concern certain quantiles, although there are many possible approaches.

In the Bayes Linear case only the two numbers, which give the expectation and variance of each contribution, are required. Often, the expectations are assumed equal to zero, which states that the modeller has symmetric beliefs about the deficiencies of the model. Obtaining an assessment of the variance of each contribution requires more thought, but various methods can be used including: assessing the standard deviation directly, or assuming approximate distributions combined with beliefs about quantiles.

Multivariate Output

If the model produces several outputs that can be compared to observed data, then we can choose to consider each of the r outputs separately and assess the model discrepancy for each individual output as described in the previous sections.

If the joint structure of the outputs is considered important, as is most likely the case, a more rigorous approach is to assess the full multivariate model discrepancy. In this case y, $f(x)$, d and $\mathrm{E}[d]$ are all vectors of length r and $\mathrm{Var}[d]$ is an $r \times r$ covariance matrix. Assessing either the full multivariate distribution for d (in the fully Bayesian case) or $\mathrm{E}[d]$ and $\mathrm{Var}[d]$ (in the Bayes linear case), can be a complex task. However, as outlined above, consideration of improvements to the model can suggest a natural correlation structure for d, which when combined with consideration of the structures of the physical processes described by the model can suggest corresponding low dimension parameterisations for the expectation $\mathrm{E}[d]$ and covariance matrix $\mathrm{Var}[d]$. Then only a small number of parameter values need be assessed which can be done directly, by using the techniques described in the previous sections or by use of purpose built elicitation tools. For discussion and examples of possible model discrepancy structures and parameterisations see *DiscStructuredMD* and for an example of an elicitation tool see Vernon et al (2010) or Bower et al (2009).

8.88.3 Additional Comments

Note that several of the formal methods of assessing the model discrepancy discussed in page *DiscFormalAssessMD* involve the specification of either prior distributions for d, or expectations and variances of d. These prior specifications would most likely be given using the methods outlined in this page.

8.88.4 References

Vernon, I., Goldstein, M., and Bower, R. (2010), "Galaxy Formation: a Bayesian Uncertainty Analysis," MUCM Technical Report 10/03

Bower, R., Vernon, I., Goldstein, M., et al. (2009), "The Parameter Space of Galaxy Formation," MUCM Technical Report 10/02,

8.89 Discussion: Factorial design

8.89.1 Description and Background

Several toolkit operations call for a set of points to be specified in the *simulator's* space of inputs, and the choice of such a set is called a design. Factorial designs refers to general class of design in which each (univarate) input, or factor, has a prespecified set of levels. A design which consists of all combinations of factor levels is called a *full factorial design*. For instance, if there are two factors, the first with 3 levels and the second with 2 levels, then the full factorial design has 6 points. Another important case is where d factors all have k levels, so that the full factorial design has k^d points. If the design is a subset of a full factorial design then it is sometimes refered to as a *fraction* (fraction of a full factorial).

A full factorial design makes a suitable *candidate set* of designs points, that is to say a large set out of which one may select a subset for the actual design used to build an emulator. In this sense factorial desiogn is a generic term and covers most designs. An exception is where points are selected at random to a certain precision.

1. Regular fractions. In this case every input (factor) takes k levels and k is a power of a prime number. In the k^d case the fractions would typically have $n = k^{d-r}$, for a suitable integer r.
2. Irregular fractions. These preserve some of the properties of regular fractions but are typically a one-off, or in small families. One of the most famous is the 12-point Plackett-Burman design, for up to 11 factors, each with 2 levels.
3. Response surface designs. These are more adventurous that regular fractions, but may have nice symmetry properties. For example for $d = 3$ one may take a 2^3 full factorial with points $(\pm 1, \pm 1, \pm 1)$ and add "star" points on the axes at $(\pm c, 0, 0), (0, \pm c, 0), (0, 0, \pm c)$. Designs built from two or more types are called *composite*.
4. *Screening* design. The classical versions of these design have links to group screening (e.g. pooling blood samples for fast elimination of disease free patients) and search problems such as binary search in computer science and information theory and games (e.g. find the bad penny out of twelve with a minimal number of balance weighings). Highly fractionated factorial designs are useful, though typically only have two or three levels. The Morris method (see the procedure page *ProcMorris*) is a kind of hybrid of factorial design and a space filling design.

8.89.2 Discussion

Factorial design has typically been used, historically, to fit polynomial models and some general principles are established in this context.

1. A factor must have at least k levels to model a polynomial up to degree $k-1$ in that factor.
2. Avoid aliasing: make sure that all the terms in the models one may be interested in can be estimated at the same time. Technically, aliasing means that two polynomials agree up to a constant on the design and therefore cannot be distinguished by the design; *e.g.* with the 2^{3-1} fraction $(-1,-1,-1),(1,1,-1),(1,-1,1),(-1,1,1)$ the terms x_1 and x_2x_3 are aliased.
3. Blocking. There may be a *blocking factor* which is not modeled explicitly but can be guarded against in the experiment. In physical experiments time and temperature are typical blocking factors
4. Replication. This is advantageous in physical experiments to obtain dependent estimates of the error variance σ^2.

8.89.3 Additional Comments, References, and Links

Boukouvalas, A., Gosling, J.P. and Maruri-Aguilar, H., An efficient screening method for computer experiments. NCRG Technical Report, Aston University (2010)

D R Cox and N Reid. The Theory Of The Design Of Experiments, CRC Press, 2000.

D-Z Du and F K Hwang. Combinatorial Group Testing and Applications, World Scientific, 1993.

G. E. P. Box, J. S. Hunter and W. G. Hunter. Statistics for Experimenters: Design, Innovation, and Discovery , 2nd Edition, Wiley, 2005.

G. Pistone, E. Riccomango, H. P. Wynn, Algebraic Statistics, CRC Press, 2001.

K S Banerjee. Weighing designs. Marcel Dekker, New York, 1975.

8.90 Discussion: Formal Assessment of Model Discrepancy

8.90.1 Description and background

We consider formal *assessment* of the *model discrepancy* term d (*ThreadVariantModelDiscrepancy*), formulated to account for the mismatch between the *simulator* f evaluated at the *best input* x^+ (*DiscBestInput*) and system value y. Unlike the informal methods described in *DiscInformalAssessMD*, the key ingredient here is a specification of $\text{Var}[d]$ in terms of relatively few unknown parameters φ. The aim is to learn about φ from system observations z (*DiscObservations*) and simulator runs (*ThreadCoreGP* and *ThreadCoreBL*). Likelihood and Bayesian methods are described. Assessing the distribution of d, or just its first and second order structure, necessary for a Bayes linear analysis, is crucial for the important topics of history matching, *calibration* and prediction for the real system, which will be discussed in future Toolkit releases.

8.90.2 Discussion

Our aim in this page is to describe formal methods for assessing the distribution of the model discrepancy term d using system observations z and simulator output F from n simulator runs at design inputs D. The distribution of d is often assumed to be Gaussian with $\text{E}[d]=0$ and variance matrix $\text{Var}[d]$ depending on a few parameters. The low-dimensional specification of $\text{Var}[d]$ mostly arises from a subject matter expert, such as the cosmologist in the galaxy formation study described in the first of the two examples given at the end of this page. Initial assessment of $\text{E}[d]$ is usually zero, but subsequent analysis may suggest there is a 'trend' which is attributable to variation over and above that accounted for by Var$[d]$.

Notation

We use the notation described in the discussion page on structured forms for the model discrepancy (*DiscStructuredMD*), where there is a 'location' input u (such as space-time) which indexes simulator outputs as $f(u, x)$ and corresponding system values $y(u)$ measured as observations $z(u)$ with model discrepancies $d(u)$.

Suppose the system is observed at k locations $u_1, \ldots, u_k$. Denote the corresponding observations by $z = (z_1, \ldots, z_k)$ and the measurement errors by $\epsilon = (\epsilon_1, \ldots, \epsilon_k)$. Denote by $\Sigma_{ij}(\varphi)$ the $r \times r$ covariance matrix Cov$[d(u_i), d(u_j)]$ for any pair of locations u_i and u_j in the set U of allowable values of u. The number of unknown parameters in φ is assumed to be small compared to the number rk of scalar observations z_{ij}. Denote by $\Sigma_d(\varphi)$ the $rk \times rk$ variance matrix of the rk-vector d, where $[\Sigma_d(\varphi)]_{ij} = \Sigma_{ij}(\varphi)$.

In the galaxy formation example, described below, $r = 1, k = 11$ and $\varphi = (a, b, c)$, so that $\Sigma_d(\varphi)$ is an 11×11 matrix. Initially, the cosmologist gave specific values for a, b and c, but subsequently gave ranges. There are 11 observations in this example, so inference about a, b and c is likely to be imprecise, unless prior information about them is precise and reliable.

In the spot welding example, $r = 1$ and there are 10 replicate observations at each of the $k = 12 = 2 \times 3 \times 2$ locations. A simple form of separable covariance is

$$\text{Cov}[d(l, c, g), d(l', c', g')] = \sigma^2 \exp(-(\theta_l(l - l')^2 + \theta_c(c - c')^2 + \theta_g(g - g')^2))$$

Here, $\varphi = (\sigma^2, \theta_l, \theta_c, \theta_g)$ and there are 48 observations to estimate these four parameters. In fact, the replication level of 10 observations per location suggests that simultaneous estimates of both φ and the measurement error variance σ_ϵ^2 should be quite precise.

Inference for φ

We now consider how to estimate the model discrepancy variance matrix $\Sigma_d(\varphi)$ using an *emulator* and system observations z.

We start by choosing likelihood as a basis for inference about φ. The likelihood $l(\varphi)$ for φ can be computed as

$$l(\varphi) \propto \int p(z|D, F, \varphi, x^+) p(x^+) dx^+$$

where $p(x^+)$ is the prior distribution for x^+. The form of the integrand follows because of the separation between F and z given $f(x^+)$ due to the strong independence property between discrepancy d and (f, x^+). The first distribution in the integrand may also be interpreted as a joint likelihood function for φ and x^+. Integration over x^+ hides the potential for this joint likelihood surface to be multimodal, as there will often be fits to the observations for some choices of x^+ with small variance and low correlation across outputs and other fits with large variance and high correlation across outputs. However, we assume there is a prior distribution for x^+, so that $l(\varphi)$ is the likelihood for φ.

The expectation and variance of the first distribution in the integrand can be computed as

$$\text{E}[z|D, F, \varphi, x^+] = \text{E}[z|D, F, x^+] = \mu(x^+)$$

and

$$\text{Var}[z|D, F, \varphi, x^+] = \Sigma(x^+) + \Sigma_d(\varphi) + \Sigma_\epsilon$$

where $\mu(x)$ and $\Sigma(x)$ are the emulator mean and emulator variance matrix at input x and Σ_ϵ is the measurement error variance matrix. For simplicity, we typically assume a Gaussian distribution for $p(z|D, F, \varphi, x^+)$, but robustness of inference to other distributions may be considered.

The integral in the expression for :math:‘ l(varphi)‘, which gives the likelihood for any particular value of φ, is computed using numerical integration or by simulating from the prior distribution $p(x^+)$ for x^+. We can then proceed to compute the maximum likelihood estimate $\hat{\varphi}$ and confidence regions for φ using the Hessian of the log-likelihood function at $\hat{\varphi}$. The maximum likelihood estimate of $\Sigma_d(\varphi)$ is $\Sigma_d(\hat{\varphi})$. Edwards, A. W. F. (1972) gives an interesting account of likelihood.

If we are prepared to quantify our prior information about φ (for example, using the considerations of the discussion page on expert assessment (*DiscExpertAssessMD*)) in terms of a prior distribution, then we may base inferences on its posterior distribution, computed using Bayes theorem in the usual way.

Bayes linear inference for φ proceeds as follows. (i) Simulation to derive mean and covariance structures between z and x^+, which are used to identify the Bayes linear assessment $\hat{x}$ for x^+ adjusted by z; (ii) evaluation of the hat run $\hat{f} = f(\hat{x})$, as in Goldstein, M. and Rougier, J. C. (2006); (iii) simulation to assess the mean, variance and covariance structures across the squared components of the difference $z - \hat{f}$ and the components of φ, to carry out the corresponding Bayes linear update for φ.

8.90.3 Additional comments and examples

It should be noted that when prediction for the real system and calibration are considered in future releases of the toolkit, it will be necessary to account for uncertainty in φ in the overall uncertainty of these procedures.

Galaxy formation

Goldstein, M. and Vernon, I. (2009) consider the galaxy formation model ‘Galform’ which simulates two outputs, the b_j and K band luminosity functions. The b_j band gives numbers of young galaxies s per unit volume of different luminosities, while the K band describes the number of old galaxies l. The authors consider 11 representative outputs, 6 from the b_j band and 5 from the K band. Here, $u = (A, \lambda)$ is age A and luminosity λ and $y(A, \lambda)$ is count of age A galaxies of luminosity λ per unit volume of space. The authors carried out a careful elicitation process with the cosmologists for $\log y$ and specified a covariance Cov$[d(A_i, \lambda_l), d(A_j, \lambda_k)]$ between $d(A_i, \lambda_l)$ and $d(A_j, \lambda_k)$ of the form

$$a \begin{bmatrix} 1 & b & .. & c & .. & c \\ b & 1 & .. & c & . & c \\ : & : & : & : & : & : \\ c & .. & c & 1 & b & .. \\ c & .. & c & b & 1 & .. \\ : & : & : & : & : & : \end{bmatrix}$$

for specified values of the overall variance a, the correlation within bands b and the correlation between bands c. The input vector x has eight components.

Spot welding

Higdon, D., Kennedy, M., Cavendish, J. C., Cafeo, J. A., and Ryne, R. D. (2004) consider a model for spot welding which simulates spot weld nugget diameter for different combinations of load and current applied to two metal sheets, and gauge is the thickness of the two sheets. Here, $u = (l, c, g)$ represents load l, current c and gauge g and $y(l, c, g)$ is the weld diameter when load l and current c are applied to sheets of gauge g at the $12 = 2 \times 3 \times 2$ combinations. Moreover, there is system replication of 10 observations for each of the 12 system combinations. The authors specify a Gaussian process for d over (l, c, g) combinations, using a separable covariance structure. There is one scalar input.

8.90.4 References

Edwards, A. W. F. (1972), “Likelihood”, Cambridge (expanded edition, 1992, Johns Hopkins University Press, Baltimore): Cambridge University Press.

Goldstein, M. and Rougier, J. C. (2006), “Bayes linear calibrated prediction for complex systems”, *Journal of the American Statistical Association*, 101, 1132-1143.

Goldstein, M. and Vernon, I. (2009), “Bayes linear analysis of imprecision in computer models, with application to understanding the Universe”, in 6th International Symposium on Imprecise Probability: Theories and Applications.

Higdon, D., Kennedy, M., Cavendish, J. C., Cafeo, J. A., and Ryne, R. D. (2004), “Combining field data and computer simulations for calibration and prediction”, *SIAM Journal on Scientific Computing*, 26, 448–466.

8.91 Discussion: Forms of GP-Based Emulators

8.91.1 Description and Background

In the fully *Bayesian* approach to *emulating* the output(s) of a *simulator*, the emulation is based on the use of a *Gaussian process* (GP) to represent the simulator output as a function of its inputs. However, the underlying model represents the output as a GP *conditional* on some *hyperparameters*. This means that a fully Bayesian (GP-based) emulator in the toolkit has two parts. The parts themselves can have a variety of forms, which we discuss here.

8.91.2 Discussion

An emulator in the fully Bayesian approach is a full probability specification for the output(s) of the simulator as a function of its inputs, that has been trained on a *training sample*. Formally, the emulator is the Bayesian posterior distribution of the simulator output function $f(\cdot)$. Within the toolkit, a GP-based emulator specifies this posterior distribution in two parts. The first part is the posterior distribution of $f(\cdot)$ conditional on the hyperparameters. The second is a posterior specification for the hyperparameters.

First part

In the simplest form, the first part is a GP. It is defined by specifying

- The hyperparameters θ upon which the GP is conditioned
- The posterior mean function $m^*(\cdot)$ and covariance function $v^*(\cdot,\cdot)$, which will be functions of (some or all of) the hyperparameters in θ

However, in some situations it will be possible to integrate out some of the full set of hyperparameters, in which case the conditional distribution may be, instead of a GP, a *t process*. The definition now specifies

- The hyperparameters θ upon which the t process is conditioned
- The posterior degrees of freedom b^*, mean function $m^*(\cdot)$ and covariance function $v^*(\cdot,\cdot)$, which will be functions of (some or all of) the hyperparameters in θ

Second part

The full probabilistic specification is now completed by giving the posterior distribution $\pi_\theta(\theta)$ for the hyperparameters on which the first part is conditioned. The second part could consist simply of this posterior distribution. However, $\pi_\theta(\cdot)$ is not generally a simple distribution, and in particular not a member of any of the standard families of distributions that are widely used and understood in statistics. For computational reasons, we therefore augment this abstract statement of the posterior distribution with one or more specific values of θ designed to provide a discrete representation of this distribution.

We can denote these sample values of θ by $\theta^{(j)}$, for $j = 1, 2, \ldots, s$, where s is the number of hyperparameter sets provided in this second part. At one extreme, s may equal just 1, so that we are using a single point value for θ. Clearly, in this case we have a representative value (that should be chosen as a "best" value in some sense), but the representation does not give any idea of posterior uncertainty about θ. Nevertheless, this simple form is widely used, particularly when it is believed that accounting for uncertainty in θ is unimportant in the context of the uncertainty expressed in the first part.

At the other extreme, we may have a very large random sample which will provide a full and representative coverage of the range of uncertainty about θ expressed in $\pi_\theta(\cdot)$. In this case, the sample may comprise s independent draws from $\pi_\theta(\theta)$, or more usually is a Markov chain Monte Carlo (MCMC) sample (the members of which will be correlated but should still provide a representative coverage of the posterior distribution).

The set of θ values provided may be fixed (and in particular this will be the case when $s = 1$), or it may be possible to increase the size of the sample.

8.91.3 Additional Comments

The way in which the sample of $\theta^{(j)}$ values is used for computation of specific tasks, and in particular the way in which the number s of θ values is matched to the number required for the computation, is considered in the discussion page on Monte Carlo estimation, sample sizes and emulator hyperparameter sets (*DiscMonteCarlo*).

8.92 Discussion: The Gaussian assumption

8.92.1 Description and Background

The fully *Bayesian* methods in *MUCM* are based on the use of a *Gaussian process emulator*. In contrast to the *Bayes linear* methods, this adds an assumption of a Gaussian (i.e. normal) distribution to represent uncertainty about the *simulator*. Whilst most of this discussion concerns fully Bayesian emulators, some aspects are relevant also to the Baykes linear approach.

8.92.2 Discussion

This discussion looks at why the Gaussian assumption is made, when it might be unrealistic in practice, and what might be done in that case. Following these comments, which are from the perspective of the fully Bayesian approach, some remarks are made about analogous issues in the Bayes linear approach.

Why the Gaussian assumption

The fully Bayesian approach to statistics involves specifications of uncertainty that are complete probability distributions. Thus, a fully Bayesian emulator must make predictions of simulator outputs in the form of (joint) probability distributions. In the MUCM toolkit, an emulator is a Gaussian process (conditional on any *hyperparameters* that might be in the mean and variance functions). Using a Gaussian process implies an assumption that uncertainty about simulator outputs may be represented (conditionally) by normal (also known as Gaussian) distributions.

In principle, the fully Bayesian approach could involve emulators based on any other kind of joint probability distributions, but the Gaussian process is by far the simplest form of probability distribution that one could use. It might be technically feasible to develop tools based on other distributional assumptions, but those tools would certainly end up being much more complex. So the MUCM toolkit does not attempt to address any other kind of specification.

When the assumption might not be realistic

First, it should be said that the assumption becomes unimportant if we are able to make enough runs of the simulator in order to produce a very accurate emulator. The point is that if there is very little uncertainty in the emulator predictions, then the precise nature of the probability distribution that we use to represent that uncertainty does not matter. So it is not necessary to question the Gaussian assumption in these circumstances.

In practice, however, the MUCM approach has been developed to work with computationally intensive simulators, for which we cannot make arbitrarily large numbers of runs. So we will often be in the situation where the Gaussian assumption is not irrelevant.

Probably the most important context where the assumption may be unrealistic is when the simulator output being emulated has a restricted range of possible values. For instance, many simulator outputs are necessarily positive. Suppose that the emulator estimates such an output (for some given input configuration) with a mean of 10 and a standard deviation of 10, then the Gaussian assumption implies that the emulator gives a nontrivial probability that the true output will be negative. Then the emulator is clearly making unrealistic statements, and in such a situation the Gaussian assumption would be inappropriate.

(Incidentally, we can see here why the assumption can be ignored when we are able to achieve high emulator accuracy, because that means the standard deviation of any prediction will always be small. Then the existence of restrictions on the range of the output will not be a problem.)

What to do if the assumption is unrealistic

We have seen that the Gaussian assumption becomes unrealistic if the emulator gives a nontrivial probability to the output lying outside its possible range, and that this may arise if the emulator has sufficient uncertainty. The simplest remedy for this is one that is widely used to deal with assumptions of normality in other areas of Statistics, namely to transform the variable.

Suppose, for example, that our simulator output is the amount of water flowing in a stream. Then it is clear that this output must be positive. If we were simulating the flow in a river, then we might find that the output over all plausible configurations of the simulator inputs does not get close to zero, and then the Gaussian assumption would not give cause for concern. In the case of a stream, however, there may be some input configurations in which the flow is large and others (e.g. drought conditions) that give very low flow. Then we might indeed meet the problem of unrealistic predictions. A solution is to define the output not to be the flow itself but the logarithm of the flow. The log-flow output is not constrained, and so the Gaussian assumption should be valid for the log-flow when it is not acceptable for the flow.

The log transformation is widely used in statistics to make a normality assumption acceptable for a variable that must be positive and which has sufficiently large uncertainty.

Notice, however, that now the emulator makes predictions about log-flow, whereas we will generally wish to predict the actual output, i.e. flow. If the emulator prediction for log-flow (for some given input configuration) has a normal distribution with mean m and variance v, for instance, then flow has mean $\exp(m + v/2)$. In general, if we transform the output in order to make the Gaussian assumption acceptable, then the question arises of how to construct appropriate inferences about the untransformed variable. In the fully Bayesian approach this is a technical question that can readily be addressed (although it may involve substantial extra computations).

Details of how to deal with transformed outputs can be found in the the procedure page for transforming outputs (*ProcOutputTransformation*).

Bayes linear methods

There is a Bayes linear formulation in which simulator outputs are characterised in terms of their means, variances and covariances, modelling these in similar ways to a full Bayesian analysis. The basic Bayes linear updating formulae will then produce analogous results to the full Bayesian posterior means, variances and covariances. Probability distributions are not required in the Bayes linear approach, so the specification of means, variances and covariances will not be extended by making the Gaussian assumption. From the Bayes linear viewpoint, this analysis is meaningful without that assumption. From the perspective of the full Bayesian approach, however, the Gaussian assumption is necessary, and results cannot be legitimately obtained without asserting probability distributions.

Within the Bayes linear approach we might also prefer to build an emulator for the logarithm or some other transformation of the output of interest, rather than the output itself. Then the question of making statements about the untransformed output based on an emulator of the transformed output is more complex because the mean and variance of the original output are not implied by the mean and variance of the transformed output. We therefore have to build a joint belief specification for both the output and the transformed output, which poses technical difficulties.

8.92.3 Additional Comments

This discussion is relevant to all MUCM methods, since they all involve emulation. See in particular the core threads *ThreadCoreGP* and *ThreadCoreBL*.

8.93 Discussion: Implausibility Cutoffs

8.93.1 Description and Background

As introduced in *ThreadGenericHistoryMatching*, an integral part of the *history matching* process is the imposing of cutoffs on the various *implausibility measures* introduced in *AltImplausibilityMeasure*. This page discusses the relevant considerations involved in choosing the cutoffs. The notation used is the same as that defined in *ThreadGenericHistoryMatching*. Here we use the term model synonymously with the term *simulator*.

8.93.2 Discussion

History matching attempts to identify the set $\mathcal{X}$ of all inputs that would give rise to acceptable matches between model outputs and observed data. This is achieved by imposing cutoffs on implausibility measures in order to discard inputs x that are highly unlikely to belong to $\mathcal{X}$. The specific values used for these cutoffs is therefore important, and several factors must be considered before choosing them.

Univariate Cutoffs

Consider the univariate implausibility measure $I_{(i)}(x)$ corresponding to the i, introduced in *AltImplausibilityMeasure*:

$$I^2_{(i)}(x) = \frac{(\mathrm{E}[f_i(x)] - z_i)^2}{\mathrm{Var}[\mathrm{E}[f_i(x)] - z_i]} = \frac{(\mathrm{E}[f_i(x)] - z_i)^2}{\mathrm{Var}[f_i(x)] + \mathrm{Var}[d_i] + \mathrm{Var}[e_i]}$$

where the second equality follows from the definition of the best input approach (see *DiscBestInput* for details). We are to impose a cutoff c such that values of x that satisfy:

$$I_{(i)}(x) \leq c$$

are to be analysed further, and all other values of x are discarded. This defines a new sub-volume of the input space that we refer to as the non-implausible volume. Determining a reasonable value for the cutoff c can be achieved using certain unimodality arguments, which are employed as follows.

Regarding the size of the individual univariate implausibility measure $I_{(i)}(x)$, we again consider x as a candidate for the best input x^+. If we then make the fairly weak assumption that the appropriate distribution of $(\mathrm{E}[f_i(x)] - z)$, with $x = x^+$ rule (Pukelsheim, 1994) which implies quite generally that $I_{(i)}(x) \leq 3$ with probability greater than 0.95, even if the distribution is heavily asymmetric. A value of $I_{(i)}(x)$ greater than a cutoff of $c = 3$ would suggest that the input x could be discarded.

Consideration of the fraction of input space that is removed for different choices of c may alter this value. For example, if we find that we can remove a sufficiently large percentage of the input space with a more conservative choice of say $c = 4$, then we may adopt this value instead. In addition, we may also perform various diagnostics to check that we are not discarding any acceptable runs (Vernon, 2010).

Multivariate Cutoffs

We can use the above univariate cutoff c as a guide to determine an equivalent cutoff c_M for the maximum implausibility measure such that we discard all x that do not satisfy

$$I_M(x) \leq c_M$$

If we are dealing with a small number of highly correlated outputs, then this will be similar to the univariate case and we can use Pukelsheim's 3 sigma rule and choose values for c_M that are similar to or slightly larger than $c = 3$ (e.g. $c_M = 3.2, 3.5$ say). If there are a large number of outputs, and we believe them to be mostly uncorrelated then a higher value must be chosen (e.g. $c_M = 4$, 4.5 or 5 say) depending on the fraction of space cut out. If we want to be more precise, we can make further assumptions as to the shape of the individual distributions that are used in the implausibility measures: for example we can assume they are normally distributed, and look up suitable tables for the maximum of a collection of independent or even correlated normals at certain conservative significance levels (we would recommend 0.99 or higher). Similar considerations can be used to generate sensible cutoffs for the second and third maximum implausibility measures. See *DiscInformalAssessMD* for similar discussions regarding cutoffs, and Vernon, 2010 for examples of their use.

Consider the full multivariate implausibility measure $I_{MV}(x)$ introduced in *AltImplausibilityMeasure*:

$$I_{MV}(x) = (z - \mathrm{E}[f(x)])^T (\mathrm{Var}[f(x)] + \mathrm{Var}[d] + \mathrm{Var}[e])^{-1} (z - \mathrm{E}[f(x)])$$

Choosing a suitable cutoff c_{MV} for $I_{MV}(x)$ is more complicated. As a simple heuristic, we might choose to compare $I_{MV}(x)$ with the upper critical value of a χ^2-distribution with degrees-of-freedom equal to the number of outputs considered. This should then be combined with considerations of percentage space cutout along with diagnostics as discussed above.

8.93.3 Additional Comments

In order to link the different cutoffs c, c_M and c_{MV} rigorously, we would of course have to make full multivariate distributional assumptions over each of the relevant output quantities. This is a level of detail that is in many cases not necessary, provided relatively conservative choices for each of the cutoffs are made.

The history matching process involves applying the above implausibility cutoffs iteratively, as is described in *ThreadGenericHistoryMatching* and discussed in detail in *DiscIterativeRefocussing*. The implausibility measures are simple and very fast to evaluate (as they only rely on evaluations of the emulator), and hence the application of the cutoffs is tractable even for testing large numbers of input points.

8.93.4 References

Pukelsheim, F. (1994). “The three sigma rule.” *The American Statistician*, 48: 88–91.

Vernon, I., Goldstein, M., and Bower, R. (2010), “Galaxy Formation: a Bayesian Uncertainty Analysis,” MUCM Technical Report 10/03

8.94 Discussion: Informal Assessment of Model Discrepancy

8.94.1 Description and background

We consider informal *assessment* of the *model discrepancy* term d (*ThreadVariantModelDiscrepancy*), formulated to account for the mismatch between the *simulator* f evaluated at the *best input* x^+ (*DiscBestInput*) and real system value y (*DefModelDiscrepancy*). Such informal assessment will be described mainly in terms of estimating $\textrm{Var}[d]$ based on system observations z (*DiscObservations*) and simulator runs (*ThreadCoreGP* and *ThreadCoreBL*).

8.94.2 Discussion

Our aim in this page is to assess informally the variance $\textrm{Var}[d]$ of the discrepancy term $d = y - f(x^+)$ using system observations z and simulator output F from n simulator runs at design inputs D. Our initial assessment of $\textrm{E}[d]$ is usually zero, but subsequent analysis may suggest there is a 'trend' attributable to variation over and above that accounted for by $\textrm{Var}[d]$.

It should be noted that it is only helpful to assess discrepancy by matching simulated output to observational data when we have many observations, as otherwise the effects of over-fitting will dominate. Also, while we are focussing here on informal model discrepancy assessment methods, these should be considered in combination with expert assessment, as discussed in *DiscExpertAssessMD*.

The basic idea is to estimate the variance $\textrm{Var}[d]$ using differences $z-f(x)$ evaluated over a carefully selected collection of simulator runs. We consider two situations: when the simulator is fast to run and when it is slow to run. In the fast case, we consider a further dichotomy: the simulator is either a black box or an open box, where we have access to the internal code and the equations governing the mathematical model. In the open box case, we decompose model discrepancies into internal and external model discrepancies.

Notation

We use the notation described in the discussion page on structured forms for model discrepancy (*DiscStructuredMD*), where there is a 'location' input u (such as space-time) which indexes simulator outputs as $f(u, x)$ and corresponding system values $y(u)$ measured as observations $z(u)$ with model discrepancies $d(u)$.

Suppose the system is observed at k 'locations' $u_1, \ldots, u_k$. Denote the corresponding observations by $z_1, \ldots, z_k$, where $z_i \equiv z(u_i)$, the measurement errors by $\epsilon_1, \ldots, \epsilon_k$, where $\epsilon_i \equiv \epsilon(u_i)$, the system values by $y_1, \ldots, y_k$, where $y_i \equiv y(u_i)$, the simulator values at input x by $f_1(x), \ldots, f_k(x)$, where $f_i(x) \equiv f(u_i, x)$, and the model discrepancies by $d_1, \ldots, d_k$, where $d_i \equiv d(u_i)$. Then $y_i = f_i(x) + d_i$ and $z_i = y_i + \epsilon_i$ so that $z_i = f_i(x) + d_i + \epsilon_i$ for $i = 1, \ldots, k$.

Fast black box simulators

In this section, we consider black box simulators with a fast run-time for every input combination (u, x), allowing for a detailed comparison of the system observations $z_1, \ldots, z_k$ to the $n \times r \times k$ array of simulator outputs $f_i(x_j) = f(u_i, x_j)$ at inputs $x_1, \ldots, x_n$ and locations $u_1, \ldots, u_k$, where the number of simulator runs n is very large compared to the number of components of x. We choose a space-filling design D over the usually rectangular input space $\mathcal{X}$; for example, a Latin hypercube design (*ProcLHC*).

The aim now is to choose those inputs from D with outputs which are 'close' to the system observations according to some sensible criterion, for which there are many possibilities. One such possibility, the one we choose here to illustrate the process, is to evaluate the so-called 'implausibility function'

$$I(x) = \max_{1 \leq i \leq k} (z_i - f_i(x))^{\textrm{T}} {\Sigma_i}^{-1} (z_i - f_i(x))$$

at input x with output $f_i(x)$, where $\Sigma_i = \Sigma_\epsilon + \Sigma_{d_i}$ with Σ_ϵ denoting the variance matrix of the measurement errors (assumed to be known and the same for each ϵ_i) and Σ_{d_i} is the variance matrix of the discrepancy d_i. Note that it would be possible to build a full multivariate implausibility function taking into account all of the correlations across discrepancies at different locations but this would require a much more detailed level of prior specification, so that we often prefer to use the simpler form above.

We can evaluate $I(x)$ provided the Σ_{d_i} are known. When they are unknown, the case we are considering here, we propose a modification based on setting the $\Sigma_{d_i} = 0$ in the implausibility function $I(x)$, defined above.

The key idea is to use the implausibility concept to 'rule out' any input x for which this modified $I(x)$ is 'too large', according to some suitable cutoff C determined; for example, by following the development detailed in Goldstein, M., Seheult, A. and Vernon, I. (2010). The distributional form of $I(x)$ when $x = x^+$ can be simply derived and computed, assuming that the k components in the maximum are either independent chi-squared random quantities with r degrees-of-freedom, or are completely dependent when they are each set equal to the same chi-squared random quantity with r degrees-of-freedom. Of course, while none of these distributional assumptions will be true, the values of C should provide a useful pragmatic 'yardstick'.

We now select x^+ candidates to be that subset D_J of the rows of D corresponding to those inputs x_j for which $I(x_j) \leq C$. Denote the corresponding outputs by F_J. Note that D_J could be empty, suggesting that model discrepancy can be large. In practice, we increase the value of C to get a non-empty set of x^+ candidates.

To simplify the discussion, we focus on the univariate case $r = 1$ so that the variance matrices in the implausibilty function are all scalars and write Σ_i as $\sigma_i^2 = \sigma_\epsilon^2 + \sigma_{d_i}^2$. We now choose σ_{d_i} so that

$$\max_{j \in J} \left| \frac{z_i - f_i(x_j)}{\sigma_i} \right| \leq 3$$

although we could, for example, use another choice criterion, such as using one element of J which fits well across all u. Note that the choice of σ_{d_i} can be zero, whichever criterion we opt to use. On the other hand, a large discrepancy standard deviation σ_{d_i} indicates that the simulator may fail to predict well for reasons not explained by measurement error. At this point, we could evaluate the implausibility function using the new value of σ_{d_i}, which would be informative about the number of runs that are now close to the engineered value of $C = 3$ and their location in input space, which might be of interest. Note that an assessment of zero variance for a model discrepancy term does not suggest that the model is perfect but simply reflects the situation that we can find good enough fits from our modelling to the data that we are not forced to introduce such a term, so that our views as to the value of introducing model discrepancy can only be formed from expert scientific judgements as to model limitations, as described in *DiscExpertAssessMD*.

Slow simulators

Informal assessment of model discrepancy for a slow simulator is similar to that for a fast simulator, except we replace the simulator by a relatively fast *emulator* (*ThreadCoreGP* and *ThreadCoreBL*). In the univariate case $r = 1$, we replace the definition of the implausibility function in by

$$I(x) = \max_{1 \leq i \leq k} \left| \frac{z_i - \mathrm{E}[f_i(x)]}{\sigma_i(x)} \right|$$

where $\mathrm{E}[f_i(x)]$ denotes the emulator mean for input x at location u_i and $\sigma_i^2(x)$ is the sum of three variances: measurement error variance, model discrepancy variance and the emulator variance $\mathrm{Var}[f_i(x)]$ at x.

Since emulator run time will be fast compared to that for the simulator it emulates, we can evaluate it at many inputs (as we did for fast simulators) to help determine implausible inputs. As we did for a fast simulator, we set the discrepancy variance contribution to $\sigma_i^2(x)$ equal to zero to help identify some 'non-implausible' inputs with which to assess discrepancy standard deviation, using a procedure analogous to that for fast simulators.

Fast open box simulators

As stated above, a fast open box simulator refers to a situation where we have access to the internal code and the equations governing the mathematical model. In this case, we consider two components of model discrepancy: internal and external model discrepancy.

Internal discrepancy refers to intrinsic limitations to the model whose order of magnitude we will quantify using simulator output. For example, a forcing function $F(u)$, such as actual rainfall for a runoff model, may only be determined within 10%. Then we may assess the effect on model output by making a series of model evaluations with varying values of the forcing function within the specified limits. We may implement this by specifying a distribution for the unknown 'true values' $F_1, \dots, F_k$ of the forcing function at locations $u_1, \dots, u_k$ that reflects our beliefs and knowledge about their uncertain values, such as the 10% example above. We then sample repeatedly from this distribution and evaluate the simulator output for each forcing function sample. The associated variation in simulator output allows us to assess the internal discrepancy variation due to forcing function uncertainty. Internal discrepancy variation attributable to other model features, such as initial and boundary conditions, may be assessed similarly; see Goldstein, M., Seheult, A. and Vernon, I. (2010) for a detailed account. Finally, all of the internal discrepancies considered are accumulated into an approximate overall internal discrepancy variance. While it is informative to assess each of the individual internal discrepancy contributions, this does require a large number of model evaluations. If the model is not fast enough, or if we suspect some of the discrepancy contributions to be highly correlated, then it is often advisable to vary simultaneously the intrinsic limitations to assess the overall internal discrepancy variance, rather than treat them as if they were independent as in the approximate overall assessment described above. It should be emphasised that the overall internal discrepancy variance will be used when the values of each of the intrinsic limitation contributions (such as a forcing function) are fixed in the simulator, most likely at their original nominal values. *DiscExchangeableModels* suggests an alternative approach to forcing function uncertainty and other intrinsic limitations to the mathematical model.

We refer to those remaining aspects of model discrepancy concerning differences between the model and the system, arising from features which we cannot quantify by simulation, as external structural discrepancies. Their overall variation can now be assessed using the methods described for fast black box simulators, except that selection of x^+ candidates is improved by adding the internal discrepancy variance to the measurement error variance in the modified implausibility function.

Note that there is no hard and fast division between aspects of discrepancies that we can form a judgement on by tweaking the model and aspects that we can form a judgement on by expert assessment. Both methods can be treated in exactly the same way in determining portions of the overall discrepancy variance in this approach.

8.94.3 Additional comments

Where we simplified the discussion in the fast black box simulators section by focussing on the case $r = 1$, we can still treat the general case by stringing out the $r \times k$ output into an rk-vector and follow the $r = 1$ procedure. While this is reasonable informally, it only assesses variances, not covariances; but informal covariance assessment would require more observations than typically available.

8.94.4 References

Goldstein, M., Seheult, A. and Vernon, I. (2010), “Assessing Model Adequacy”, MUCM Technical Report 10/04.

8.95 Discussion: Iterative Refocussing

8.95.1 Description and Background

As introduced in *ThreadGenericHistoryMatching*, an integral part of the *history matching* process is the iterative reduction of input space by use of implausibility cutoffs, a technique known as refocussing. This page discusses this strategy in more detail, and describes why iterative refocussing is so useful. The notation used is the same as that defined in *ThreadGenericHistoryMatching*. Here we use the term model synonymously with the term *simulator*.

8.95.2 Discussion

We are attempting to identify the set of acceptable inputs $\mathcal{X}$, and we do this by reducing the input space in a series of iterations or *waves*. We denote the original input space as $\mathcal{X}_0$.

Refocussing: Motivation

Consider the univariate implausibility measure $I_{(i)}(x)$ corresponding to the i, introduced in *AltImplausibilityMeasures*:

$$I^2_{(i)}(x) = \frac{(\mathrm{E}[f_i(x)] - z_i)^2}{\mathrm{Var}[f_i(x)] + \mathrm{Var}[d_i] + \mathrm{Var}[e_i]}$$

If we impose a cutoff c on the original input space $\mathcal{X}_0$ such that

$$I_i(x) \leq c$$

then this defines a new volume of input space that we refer to as non-implausible after wave 1 and denote $\mathcal{X}_1$. In the first wave of the analysis $\mathcal{X}_1$ will be substantially larger than the set of interest $\mathcal{X}$, as it will contain many values of x that only satisfy the implausibility cutoff because of a substantial emulator variance $\mathrm{Var}[f_i(x)]$. If the emulator was sufficiently accurate over the whole of the input space that $\mathrm{Var}[f_i(x)]$ was small compared to the model discrepancy and the observational error variances $\mathrm{Var}[d_i]$ and $\mathrm{Var}[e_i]$, then the non-implausible volume defined by $\mathcal{X}_1$ would be comparable to $\mathcal{X}$ and the history match would be complete. However, to construct such an accurate emulator would, in most applications, require an infeasible number of runs of the model. Even if such a large number of runs were possible, it would be an extremely inefficient method: we do not need the emulator to be highly accurate in regions of the input space where the outputs of the model are clearly very different from the observed data.

This is the main motivation for the iterative approach: in each wave we design a set of runs only over the current non-implausible volume, emulate using these runs, calculate the implausibility measure and impose a cutoff to define a new (smaller) non-implausible volume. This is referred to as refocusing.

Note that although we use implausibility cutoffs to define a non-implausible sub-space $\mathcal{X}_1$, we only have an implicit expression for this sub-space and cannot generate points from it directly. We can however generate a large space filling design of points over $\mathcal{X}_0$ and discard any points that do not satisfy the implausibility cutoff and which are hence not in $\mathcal{X}_1$. We use this technique in the iterative method discussed below.

Iterative Refocussing

This method, which has been found to work in practice, can be summarised as follows. At each iteration or wave:

1. A design for a set of runs over the current non-implausible volume $\mathcal{X}_j$ is created. This is done by first generating a very large *latin hypercube*, or other space filling design, over the full input space. Each of these design points are then tested to see if they satisfy the implausibility cutoffs in every one of the previous waves: if they do then they are included in the next set of simulator runs.

2. These runs (including any non-implausible runs from previous waves) are used to construct a more accurate emulator, which is defined only over the current non-implausible volume $\mathcal{X}_j$, as it has been constructed purely from runs that are members of $\mathcal{X}_j$.
3. The implausibility measures are then reconstructed over $\mathcal{X}_j$, using the new, more accurate emulator.
4. Cutoffs are imposed on the implausibility measures and this defines a new, smaller non-implausible volume $\mathcal{X}_{j+1}$ which should satisfy $\mathcal{X} \subset \mathcal{X}_{j+1} \subset \mathcal{X}_j$.
5. Unless the stopping criteria have been reached, that is to say, unless the emulator variance $\mathrm{Var}[f(x)]$ is found to be everywhere far smaller than the combined model discrepancy variance and observational errors, or unless computational resources have been exhausted, return to step 1.

If it becomes clear from say, projected implausibility plots (Vernon, 2010), that $\mathcal{X}_j$ can be enclosed in some hypercube that is smaller than the original input space $\mathcal{X}_0$, then we would do this, and use the smaller hypercube to generate the subsequent large latin hypercubes required in step 1.

As we progress through each iteration the emulator at each wave will become more and more accurate, but will only be defined over the previous non-implausible volume given in the previous wave. The increase in accuracy will allow further reduction of the input space. The process should terminate when certain stopping criteria are achieved as discussed below. It is of course possible (even probable) that computational resources will be exhausted before the stopping criteria are satisfied. In this case analysis of the size, location and shape of the final, non-implausible region should be performed (which is often of major interest to the modeller), possiblywith a subsequent probabilistic calibration if required.

Improved Emulator Accuracy

We usually see significant improvement in emulator accuracy for the current wave, as compared to previous waves. That is to say, we see the emulator variance $\mathrm{Var}[f(x)]$ generally decreasing with increasing wave number, while the emulator itself maintains satisfactory performance (as judged by emulator *diagnostics*).

We expect this improvement in the accuracy of the emulator for several reasons. As we have reduced the size of the input space and have effectively zoomed in on a smaller part of the function, we expect the function to be smoother and to be more easily approximated by the regression terms in the emulator (see *ThreadCoreBL* for details), leading to a better fit, with reduced residual variances for all outputs. Due to the increased density of runs over $\mathcal{X}_j$ compared to previous waves, the *Gaussian process* term in the emulator, $w(x)$, (which is updated by the new runs) will be more accurate and have lower variance as the general point x will be on average closer to known evaluation inputs.

Another major improvement in the emulators comes from identifying a larger set of *active inputs*. Cutting down the input space also means that the ranges of the function outputs are reduced. Dominant inputs that previously had large effects on the outputs have likewise been constrained, and their effects lessened. This results in it being easier to identify more active inputs that were previously masked by a small number of dominant variables. Increasing the number of active inputs allows more of the function's structure to be modelled by the regression terms, and has the effect of reducing the relative size of any *nugget term*.

As the input space is reduced, it not only becomes easier to accurately emulate existing outputs but also to emulate outputs that were not considered in previous waves. Outputs may not have been considered previously because they were either difficult to emulate, or because they were not informative regarding the input space.

Stopping Criteria

The above iterative method should be terminated either when computational resources are exhausted (a likely outcome for slow models), or when it is thought that the current non-implausible volume $\mathcal{X}_j$ is sufficiently close to the acceptable set of inputs $\mathcal{X}$. An obvious case is where we find the set $\mathcal{X}_j$ to be empty: we then conclude that $\mathcal{X}$ is empty and that the model cannot provide acceptable matches to the outputs. See the end of *ThreadGenericHistoryMatching* for further discussion of this situation.

A simple sign that we are approaching $\mathcal{X}$ is when the new non-implausible region $\mathcal{X}_{j+1}$ is of comparable size or volume to the previous $\mathcal{X}_j$. A more advanced stopping criteria involves analysing the emulator variance $\mathrm{Var}[f(x)]$ over the current volume, and comparing it to the other uncertainties present, namely the model discrepancy variance $\mathrm{Var}[d]$ and the observational errors variance $\mathrm{Var}[e]$. If $\mathrm{Var}[f_i(x)]$ is significantly smaller than $\mathrm{Var}[d_i]$ and $\mathrm{Var}[e_i]$ over all of the current volume, for all i, then even if we proceed with more iterations and improve the emulator accuracy further, there will not be a significant reduction in size of the non-implausible volume.

Once the process is terminated for either of the above reasons, various visualisation techniques can be employed to help analyse the location, size and structure of the remaining non-implausible volume.

8.95.3 Additional Comments

Consider the case where in the $(j-1)$th wave, we have deemed certain inputs to be borderline implausible; that is, they lie close to the non-implausible region $\mathcal{X}_j$ and have implausibilities just higher than the cutoff. We can use the new jth wave emulator to effectively overrule this decision, if the new jth wave implausibility measure suggests these inputs are actually non-implausible at the jth wave. We can only do this for borderline points that are close to $\mathcal{X}_j$, as further away from $\mathcal{X}_j$ the new emulator has no validity. The need for such overruling is mitigated by using conservative implausibility cutoffs.

This iterative refocussing strategy is very powerful and has been used to successfully history match several complex models, e.g. oil reservoir models (Craig, 1997) and Galaxy formation models (Bower 2009, Vernon 2010). Also see *Exam1DHistoryMatch* for a simple example which highlights these ideas.

8.95.4 References

Vernon, I., Goldstein, M., and Bower, R. (2010), "Galaxy Formation: a Bayesian Uncertainty Analysis," MUCM Technical Report 10/03.

Craig, P. S., Goldstein, M., Seheult, A. H., and Smith, J. A. (1997). "Pressure matching for hydrocarbon reservoirs: a case study in the use of Bayes linear strategies for large computer experiments." In Gatsonis, C., Hodges, J. S., Kass, R. E., McCulloch, R., Rossi, P., and Singpurwalla, N. D. (eds.), Case Studies in Bayesian Statistics, volume 3, 36–93. New York: Springer-Verlag.

Bower, R., Vernon, I., Goldstein, M., et al. (2009), "The Parameter Space of Galaxy Formation," to appear in MNRAS; MUCM Technical Report 10/02.

8.96 Discussion: the Karhunen-Loeve expansion for Gaussian processes

8.96.1 Description and Background

Let $\{Z(x)\}$ be a second order stochastic process with zero mean and continuous covariance function defined on an interval $[a,b]$:

$$R(s,t) = Cov(Z(s), Z(t)) = E(Z(s)Z(t)).$$

The Karhunen-Loeve (K-L) expansion is a commonly used result of Mercer's theorem and Reproducing Kernel Hilbert Space (RKHS).

Using K-L expansion of the covariance function $R(s,t)$ is $R(s,t) = \sum_{i=0}^{\infty} \lambda_i \phi_i(s) \phi_i(t)$ where the $\{\lambda_i, i \in N\}$ and $\{\phi_i, i \in N\}$ are respectively the nonzero eigenvalues and the eigenfunctions of the following integral equation known as "Fredholm integral equation of the second kind".

$$\int_a^b R(s,t)\phi_i(t)dt = \lambda_i \phi_i(s).$$

Theory shows that the K-L expansion of $R(s,t)$ converges uniformly in both variables s and t.

The stochastic process $Z(x)$ itself can then be expressed as

$$Z(x) = \sum_{i=0}^{\infty} \lambda_i \zeta_i \phi_i(x)$$

Where we assume that the ϕ_i orthonormal functions with respect to uniform measure on $[a,b]$ and the ζ_i are independent, zero mean, unit variance Gaussian random variables.

8.96.2 Discussion: Karhunen-Loeve for the spatial (emulator) case

For the spatial case, a given covariance function between s,t represented as $R(s,t) = Cov[Z(s_1,s_2,\cdots,s_d), Z(t_1,t_2,\cdots,t_d)]$ can be approximated using the truncated K-L expansion (see below). As in the one-dimensional case, a set of eigenfunctions can be used to represent the covariance function and the process. The basic idea, for use in emulation and optimal design, is that the behaviour of the process will be captured by the first p eigenvalues and eigenfunctions.

In the case that $R(s,t)$ is separable let the eigenvalues and eigenfunctions of the correlation function in the k-th dimension be $\{\lambda_k^{(i)}\}$ and $\{\phi_k^{(i)}(x_k)\}$. Then

$$Z(x) = Z(x_1,x_2,\cdots,x_d) = \sum_i \sqrt{\lambda_i}\zeta_i \phi_i(x_1,x_2,\cdots,x_d)$$

where

$$\lambda_i = \Pi_{k=1}^d \lambda_k^{(i_k)}, \;\; \phi_i(x_1,x_2,\cdots,x_d) = \Pi_{k=1}^d \phi_k^{(i_k)}(x_k), \; i = (i_1,\cdots,i_d)$$

and the sum is over all such i. That is to say we take all products of all one dimensional expansions.

For the proposed application to the spatial sampling represented by the GP emulator we replace the problem of estimating the covariance parameters by converting the problem (approximately) to a standard Bayes optimal design problem by truncating the K-L expansion:

$$R(s,t) = \sum_{i=0}^{p} \lambda_i \phi_i(s)\phi_i(t)$$

In its simplest form take the same λ to be the (prior) variances for a Bayesian random regression obtained by the equivalent truncation of the process itself:

$$Z(x) = \sum_{i=1}^{p} \sqrt{\lambda_i}\zeta_i \phi(x)$$

Theory shows that the K-L expansion truncated to depth p has the minimum integrated mean squared approximation to the full process, among all depth p orthogonal expansions. This reduces the problem to finding the first p eigenvalues and eigenfunctions of the Fredholm equation. The truncation point p should depend on the required level of accuracy in the reconstruction of the covariance function. Because the reconstructed covariance function will typically be singular if the number of design points n is greater than p one should add a *nugget* term in this case. One way to interpret this is that all the higher frequency terms hidden in the original covariance function are absorbed into the nugget.

To summarise, the original output process in matrix terms, $Y = X\beta + Z$, where Z is from the Gaussian process, is replaced by $Y = X\beta + \Phi\gamma + \varepsilon$, where Φ is the design matrix from the $\phi_1,\cdots,\phi_p$ and $\varepsilon \sim N(0,\sigma^2)$, with small

σ, is the nugget term. The default covariance matrix for the γ parameters is to take the diagonal matrix with entries $\lambda_1, \cdots, \lambda_p$.

When the approximation is satisfactory and with a suitable nugget attached the problem is reduced to a Bayesian random regression optimal design method as in *AltOptimalCriteria*, with an appropriate criterion.

The principle at work is that the ϕ_j for larger j express the higher frequency components of the process and conversely the smaller j the lower frequency components. It is expected that this will be a useful platform for a fully adaptive version of *ProcASCM*, where the variances attached to the γ parameters should adapt to the smoothness of the process.

Finding the eigenvalues and the eigenfunctions needs numerical solution of the eigen-equations (Fredholm integral equation of the second kind). Analytical solutions only exist for some a limited range of processes. Fast numerical solutions are described in the alternatives page *AltNumericalSolutionForKarhunenLoeveExpansion*.

8.96.3 Additional Comments, References, and Links

Although the methods to approximate the K-L expansion, particularly the Haar method, are fast, the number of basis functions in the K-L expansion in d dimensions, when the product form described above is taken, can be large: p^d if we approximate to order p in every dimension. To partly avoid this blow up one can take, not every product basis function, but limit the "degree" of the approximations across all dimensions: e.g. $0 \leq \sum_{j=1}^{d} p_j \leq p$. This is the analogue of taking a polynomial basis up to a particular "total degree" as in a quadratic response surface. An alternative method is to order the products of the eigenvalues and truncate by the value. Another point to note is that for very smooth function p need not be large because the λ_i decline rapidly. Values of p from 3 to 5 are effective.

S. P. Huang, S. T. Quek and K. K. Phoon: Convergence study of the truncated Karhunen-Loeve expansion for simulation of stochastic processes, *Intl J. for Numerical Methods in Engineering*, 52:1029-1043, 2001.

8.97 Discussion: Monte Carlo estimation, sample sizes and emulator hyperparameter sets

Various procedures in this toolkit use Monte Carlo sampling to estimate quantities of interest. For example, suppose we are interested in some nonlinear function $g(.)$ of a *simulator* output $f(x)$. As $g\{f(x)\}$ is uncertain, we might choose to estimate it by its mean $\mathrm{E}[g\{f(x)\}]$, and estimate the mean itself using Monte Carlo.

We obtain a random sample $f^{(1)}(x), \ldots, f^{(N)}(x)$ from the distribution of $f(x)$, as described by the *emulator* for $f(.)$, evaluate $t_1 = g\{f^{(1)}(x)\}, \ldots, t_N = g\{f^{(N)}(x)\}$, and estimate $\mathrm{E}[g\{f(x)\}]$ by

$$\hat{\mathrm{E}}[g\{f(x)\}] = \frac{1}{N}\sum_{i=1}^{N} t_i.$$

How large should N be?

We can calculate an approximate confidence interval to assess the accuracy of our estimate. For example, an approximate 95% confidence interval is given by

$$\left(\hat{\mathrm{E}}[g\{f(x)\}] - 1.96\sqrt{\frac{\hat{\sigma}^2}{N}}, \hat{\mathrm{E}}[g\{f(x)\}] + 1.96\sqrt{\frac{\hat{\sigma}^2}{N}}\right),$$

where $\hat{\sigma}^2 = \frac{1}{N-1}\sum_{i=1}^{N}\left(t_i - \hat{\mathrm{E}}[g\{f(x)\}]\right)^2$.

Hence, we can get an indication of the effect of increasing N on the width of the confidence interval (if we ignore errors in the estimate $\hat{\sigma}^2$), and assess whether it is necessary to use a larger Monte Carlo sample size.

8.97.1 Emulator hyperparameter sets

Now suppose we wish to allow for emulator hyperparameter uncertainty when estimating $g\{f(x)\}$.

Throughout the toolkit, we represent a fully Bayesian emulator in two parts as described in the discussion page on forms of GP-based emulators (*DiscGPBasedEmulator*). The first part is the posterior (i.e. conditioned on the *training inputs* D and training outputs $f(D)$) distribution of $f(\cdot)$ conditional on hyperparameters θ, which may be a *Gaussian process* or a *t-process*. The second part is the posterior distribution of θ, represented by a set $\{\theta^{(1)},\ldots,\theta^{(s)}\}$ of emulator hyperparameter values. The above procedure is then applied as follows.

Given an independent sample of hyperparameters $\theta^{(1)},\ldots,\theta^{(N)}$, we sample $f^{(i)}(x)$ from the posterior distribution of $f(x)$ conditional on θ (the first part of the emulator) setting $\theta=\theta^{(i)}$, and proceed as before. By sampling one $f^{(i)}(x)$ for each $\theta^{(i)}$, we guarantee independence between $t_1,\ldots,t_N$ which is required for the confidence interval formula given above. (If MCMC has been used to obtain $\theta^{(1)},\ldots,\theta^{(N)}$ then the hyperparameter samples may not be independent. In this case, we suggest checking the sample $t_1,\ldots,t_N$ for autocorrelation before calculating the confidence interval).

We have seen above how to assess the sample size N required for adequate Monte Carlo computation, and for this purpose the size s of the set of emulator hyperparameter values may need increasing. Usually, the computational cost of obtaining additional hyperparameter values will be small relative to the cost of the Monte Carlo procedure itself.

If, for some reason, it is not possible or practical to obtain additional hyperparameter values, then we recommend cycling through the sample $\{\theta^{(1)},\ldots,\theta^{(s)}\}$ as we generate each of $t_1,\ldots,t_N$. Again, we can check the sample $t_1,\ldots,t_N$ for autocorrelation before calculating the confidence interval. If possible, it is worth exploring whether the distribution of $g\{f(x)\}$ (or whatever quantity is of interest) is robust to the choice of θ, for example, by estimating $\mathrm{E}[g\{f(x)\}]$ for different fixed choices of θ from the sample $\{\theta^{(1)},\ldots,\theta^{(s)}\}$.

In some cases, we may have a single estimate of θ, perhaps obtained by maximum likelihood. Monte Carlo (and alternative) methods still allow us to usefully consider simulator uncertainty about $g\{f(x)\}$, although we are now not allowing for hyperparameter uncertainty. Again, if possible we recommend testing for robustness to different choices of θ.

8.98 Discussion: The Observation Equation

8.98.1 Description and background

Observations on the real system, measurement errors and their relation to the real system are presented in the variant thread on linking models to reality using *model discrepancy* (*ThreadVariantModelDiscrepancy*). Here, we discuss observations on the real system in more detail.

Notation

In accordance with the standard toolkit notation, we denote the model *simulator* by f and its inputs by x. We denote by z an observation or measurement on a real system value y, and denote by ϵ the measurement error (*ThreadVariantModelDiscrepancy*). The notation covers replicate observations, as described in the discussion page on structured forms for the model discrepancy (*DiscStructuredMD*).

8.98.2 Discussion

Statistical assumptions

While a value for z is observed (for example, the temperature at a particular time and place on the Earth's surface), neither the real system value y (the actual temperature at that time and place) nor the measurement error ϵ is observed.

We link these random quantities z, y and ϵ using the observation equation

$$z = y + \epsilon$$

as defined in *ThreadVariantModelDiscrepancy*.

Typical statistical assumptions are that y and ϵ are independent or uncorrelated with $\text{E}[\epsilon] = 0$ and $\text{Var}[\epsilon] = \Sigma_\epsilon$.

The variance matrix Σ_ϵ is often assumed to be either completely specified (possibly from extensive experience of using the measurement process) and is the same for each observation, or have a particular simple parameterised form; for example, $\sigma_\epsilon^2 \Sigma$, where Σ is completely specified and σ_ϵ is an unknown scalar standard deviation that we can learn about from the field observations z, especially when there are replicate observations; see *DiscStructuredMD*.

Simple consequences of the statistical assumptions are $\text{E}[z] = \text{E}[y]$ and $\text{Var}[z] = \text{Var}[y] + \Sigma_\epsilon$. Thus, z is unbiased for the expected real system value but the variance of z is the measurement error variance inflated by $\text{Var}[y]$ which can be quite large, as it involves model discrepancy variance: see, *ThreadVariantModelDiscrepancy*.

Observations as functions of system values

Sometimes our observations z are known functions $g(y)$ of system values y plus measurement error. The linear case $z = Hy + \epsilon$, where H is a matrix which can either select a collection of individual components of y, or more general linear combinations, such as averages of certain components of y, is straightforward and can be dealt with using our current methodology. In this case, both the expectation and variance of z are simply expressed. Moreover, $Hy = Hf(x^+) + Hd$, which we can re-write in an obvious reformulation as $y' = f'(x^+) + d'$, so that x^+ is the still the *best input* and f' and d' are still independent (*DiscBestInput*).

The case where measurements $z = g(y) + \epsilon$ are made on a nonlinear function g can also be reformulared as $y' = f'(x^+) + d'$ with the usual best input assumptions. Thus, if we put $y' = g(y)$ and $f'(x) = g(f(x))$, we may write $z = y' + \epsilon$ and $y' = f'(x^{+'}) + d'$ with d' independent of f' and $x^{+'}$, and analysis may proceed as before. It should be noted, however, that when g is nonlinear, it can be shown that it is incoherent to simultaneously apply the best input approach to both f and f'. However, in practice, we choose the formulation which best suits our purpose.

8.98.3 Additional comments

Note that we are assuming measurement errors are additive, while in some situations they may be multiplicative; that is, $z = y\epsilon$, in which case we can try either to work directly with the multiplicative relationship or with the additive relationship on the logarithmic scale, $\log z = \log y + \log \epsilon$, with neither case being straightforward. However, note that this case is covered by the discussion above about nonlinear functions, with $g(y) = \log(y)$.

Sometimes we have replicate system observations: see *DiscStructuredMD* for a detailed account, including notation to accommodate generalised indexing such as space-time location of observations, which are regarded as control inputs to the simulator f in addition to inputs x.

8.99 Discussion: Finding the posterior mode of correlation lengths

8.99.1 Description and Background

This page discusses some details involved in the maximisation of the posterior distribution of the correlation lengths δ. The details discussed are the logarithmic transformation of the correlation lengths and the incorporation of derivatives of the posterior distribution in the optimisation process.

8.99.2 Discussion

Logarithmic transformation

Applying a logarithmic transformation to the correlation lengths can help finding the posterior mode, because it transforms a constrained optimisation problem to an unconstrained one. The transformation that can be applied is

$$\tau = 2\ln(\delta).$$

Additionally, it is common to maximise the logarithm of the posterior distribution, rather than the posterior itself. Therefore, the function that has to be maximised is

$$g(\tau) = \ln(\pi_\delta^*(\exp(\tau/2)))$$

which after substitution of $\pi_\delta^*(\delta)$ from the procedure page on building a Gaussian Process emulator for the core problem (*ProcBuildCoreGP*), we have

$$g(\tau) \propto \ln(\pi_\delta(\exp(\tau/2))) - \frac{n-q}{2}\ln(\hat{\sigma}^2) - 0.5\ln|A| - 0.5\ln|H^{\mathrm{T}}A^{-1}H|.$$

In the absence of any prior information about the correlation lengths, the prior distribution $\pi_\delta(\delta)$ can be set to a constant, in which case the entire term $\ln(\pi_\delta(\exp(\tau/2)))$ can be ignored in the optimisation.

We should also note, that the Jacobian of the transformation $\tau = 2\ln(\delta)$ is not taken into account. This implies that $\exp(g(\tau))$ cannot be strictly viewed as a probability density function, but on the other hand, this results in the mode of $g(\tau)$ being located exactly at the same position as the mode of $\pi_\delta^*(\delta)$ for $\delta = \exp(\tau/2)$.

Optimisation using derivatives

Although the function $g(\tau)$ can be optimised using a derivatives-free algorithm, the existence of closed form expressions for the two first derivatives of $g(\tau)$ can help speeding up the optimisation. In the following, we give the two first derivatives of $g(\tau)$, assuming a uniform correlation length prior (i.e. $\pi_\delta(\delta) \propto const.$), which implies that its two first derivatives are zero. In case a different prior is used for δ, its two first derivatives need to be calculated and added to the following expressions.

The first derivative of $g(\tau)$ is a $p \times 1$ vector, whose k-th element is:

$$\frac{\partial g(\tau)}{\partial \tau_k} = -\frac{1-n+q}{2}\mathrm{tr}\left[P\frac{\partial A}{\partial \tau_k}\right] - \frac{n-q}{2}\mathrm{tr}\left[R\frac{\partial A}{\partial \tau_k}\right]$$

The second derivative is a $p \times p$ matrix, whose (k,l)-th entry is

$$\frac{\partial^2 g(\tau)}{\partial \tau_l \partial \tau_k} = -\frac{1-n+q}{2}\mathrm{tr}\left[P\frac{\partial A}{\partial \tau_l \partial \tau_k} - P\frac{\partial A}{\partial \tau_l}P\frac{\partial A}{\partial \tau_k}\right] - \frac{n-q}{2}\mathrm{tr}\left[R\frac{\partial A}{\partial \tau_l \partial \tau_k} - R\frac{\partial A}{\partial \tau_l}R\frac{\partial A}{\partial \tau_k}\right]$$

The matrices P, R are defined as follows

$$P \equiv A^{-1} - A^{-1}H(H^{\mathrm{T}}A^{-1}H)^{-1}H^{\mathrm{T}}A^{-1}$$
$$R \equiv P - Pf(D)(f(D)^{\mathrm{T}}Pf(D))^{-1}f(D)^{\mathrm{T}}P$$

The derivatives of the matrix A w.r.t τ are $n \times n$ matrices, whose (i,j)-th element is

$$\left(\frac{\partial A}{\partial \tau_k}\right)_{i,j} = (A)_{i,j}\frac{(x_{k,i} - x_{k,j})^2}{e^{\tau_k}}$$

In the above equation, $(\cdot)_{i,j}$ denotes the (i,j)-th element of a matrix, and the double subscripted $x_{k,i}$ denotes the k-th input of the i-th design point. The second cross derivative of A is

$$\left(\frac{\partial^2 A}{\partial \tau_l \partial \tau_k}\right)_{i,j} = (A)_{i,j}\frac{(x_{l,i} - x_{l,j})^2}{e^{\tau_l}}\frac{(x_{k,i} - x_{k,j})^2}{e^{\tau_k}}$$

and finally,

$$\left(\frac{\partial^2 A}{\partial \tau_k^2}\right)_{i,j} = (A)_{i,j}\left[\left(\frac{(x_{k,i}-x_{k,j})^2}{e^{\tau_k}}\right)^2 - \frac{(x_{k,i}-x_{k,j})^2}{e^{\tau_k}}\right]$$

If we denote by $\hat{\tau}$ the value of τ that maximises $g(\tau)$, the posterior mode of $\pi^*_\delta(\delta)$ is found simply by the back transformation $\hat{\delta} = \exp(\hat{\tau}/2)$.

8.99.3 Additional Comments

The posterior distribution of the correlation lengths can often have multiple modes, especially in emulators with a large number of inputs. It is therefore recommended that the optimisation algorithm is run several times from different starting points, to ensure that the global maximum is found.

8.100 Discussion: Design for generating emulator realisations

8.100.1 Description and Background

The procedure page for simulating realisations of an emulator (*ProcSimulationBasedInference*) presents a method for drawing random realisations of an *emulator*. In order to do this it is necessary to sample values of the emulator's predictive distributions for a finite set of input points $x'_1, x'_2, \ldots, x'_{n'}$. We discuss here the choice of these points, the realisation design.

8.100.2 Discussion

The underlying principle is that if we rebuild the emulator, augmenting the original training data with the sampled values at the realisation design points, then the resulting emulator would be sufficiently accurate that it would have negligible uncertainty in its predictions. To create a suitable realisation design, we need to decide how many points it should have, and where in the input space those points should be.

If we consider the second decision first, suppose we have decided on n'. Then it is clear that the combined design of n original training design points and the n' realisation design points must be a good design, since good design involves getting an accurate emulator - see the alternatives page on training sample design for the core problem (*AltCoreDesign*). For instance, we might choose this design by generating random Latin hypercubes (LHCs) of n' points and then choosing the composite design (original training design plus LHC) to satisfy the maximin criterion - see the procedure for generating an optimised Latin hypercube design (*ProcOptimalLHC*).

To determine a suitable value for n' will typically involve a degree of trial and error. If we generate a design with a certain n' and the predictive variances are not all small, then we need a larger sample!

8.100.3 Additional Comments

This topic is under investigation in MUCM, and fuller advice may be available in due course.

8.101 Discussion: Reification

8.101.1 Description and Background

The most widely used technique for linking a model to reality is that of the *Best Input* approach, described in the discussion page *DiscBestInput*. While this simple approach is useful in many applications, we often require more advanced methods to describe adequately the model discrepancy. Here we describe a more careful strategy known as Reification. More details can be found in the reification theory discussion page (*DiscReificationTheory*). Note that in this page we draw a distinction between the model (the mathematical model of the system) and the *simulator* (the code that simulates this mathematical model).

In order to think about a more detailed method for specifying model discrepancy, we first discuss three classes of approximations that occur when constructing a simulator. This leads to consideration of improved versions of the model, and we then discuss where in this new framework it would be appropriate to apply a version of the best input approach. This leads us to the reification principle, modelling and application.

8.101.2 Discussion

A More Detailed Specification of Model Discrepancy

When constructing a simulator, there are three classes of approximation that result in differences between the simulator output f and the real system values y. These are

(a) The limitations of the scientific method chosen. Even if we were able to construct a model which contained all the detailed scientific knowledge we can think about with no known approximations, it would still most likely not be exactly equal to the system y.

(b) Approximations to the scientific account of (a). Often, many theoretical approximations will be made to the model; e.g. mathematical simplifications to the theory, physical processes will be ignored and simplifications to the structure of the model will be made.

(c) Approximations in the implementation of the model. When coding the simulator, often further approximations are introduced which lead to differences between the output of the code and the mathematical model in part (b). A classic example of this is the discretisation of a continuous set of differential equations; e.g. the finite grid in a climate simulator.

These issues and their consequences can be considered at different levels of complexity. A relatively simple way of expressing this situation is by using three unknown functions f, $f_{\rm theory}$ and f^+ as follows:

- We represent the limit of the scientific method model described in (a) as f^+
- We represent the current approximate theoretical model described in (b) as $f_{\rm theory}$
- We represent the current simulator described in (c) as usual as f

and we consider how much the current simulator would be changed by moving from (c) to (b) and from (b) to (a). The move from (c) to (b) (i.e. from f to $f_{\rm theory}$) is relatively straightforward to understand and is regularly discussed by modellers (for example, as the error introduced by discretisation). The changes to f introduced in the move from (b) to (a), that is from $f_{\rm theory}$ to f^+, is far more complex and requires careful consideration. One practical way of thinking about this is to consider further theoretical improvements to the model in (b). We might represent this as the improved model $f'_{\rm theory}$ which would lie between (a) and (b). Then we can consider the changes in f when moving from $f_{\rm theory}$ to f'_{theory}, and from $f'_{\rm theory}$ to f^+.

It is now interesting to ask where the best input x^+ and the actual system y might fit into this structure (see *DiscBestInput*). It can be shown (see Goldstein, M. and Rougier, J. C. (2009)) that it would place possibly far too strong a constraint on our joint beliefs if we were to insert the same x^+ (with corresponding independent model discrepancies) into all the above models, and that a minimum description allowing maximum flexibility is obtained by inserting x^+ at the top level, into what we define as the Reified model f^+, as is discussed below.

Reified Modelling

Imagine constructing a reified model f^+ which has both (i) sufficiently good physics, and (ii) sufficiently accurate solutions that we have no additional insights as to the nature of any further improvements to f^+. Therefore, the model discrepancy that links f^+ to the real system will be unstructured, for, by definition, there is no improved model that we can think of.

Therefore, we can consistently apply the best input approach to the Reified model; i.e. to f^+ and to f^+ alone, in which case

$$y = f^+(x^+, w^+) + d^+, \qquad d^+ \perp (f, f^+, x^+, w^+)$$

where w are any extra model parameters that might be introduced due to any of the considered model improvements, and we take $\perp$ to mean "independent of" (see *DiscReificationTheory* for more details).

The Reifying Principle

The Reification approach, which gives a more careful treatment of model discrepancy than the best input approach, is based around the Reification principle, which can be stated as

- The simulator f is informative for y, because f is informative for the reified model f^+

We can represent this in terms of a Bayesian Belief Network, where 'child' vertices that are strictly determined by their 'parents' are indicated with dashed lines:

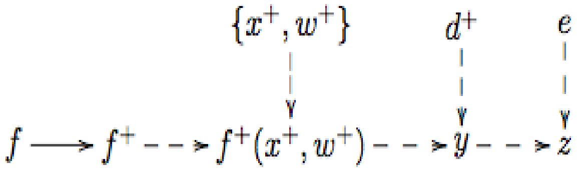


Fig. 37: **Figure 1:** Bayesian Belief Network for the Reification principle.

The reifying principle should be seen as a sensible pragmatic compromise which retains the essential tractability in linking computer evaluations and system values with system behaviour, while removing logical problems in simple treatments of discrepancy, and providing guidance for discrepancy modelling.

If we have several simulators $f_1, f_2, \ldots, f_r$, then the reifying principle suggests that we combine their information by treating each simulator as informative for the single reified form f^+.

Applying Reification

In order to apply the Reification process, we need to link the current model f, with the Reified model f^+. Often we would employ the use of an emulator to represent f, and from this construct an emulator for f^+. As introduced above, we would possibly consider emulators for intermediate models f' to resolve specified deficiencies in our modelling. This offers a formal structure to implement the methods suggested in *DiscExpertAssessMD* for consideration of model discrepancy and helps bridge the gap between f and f^+. The details of this process, and further discussion of these issues can be found in *DiscReificationTheory*.

It should be noted that although Reification can sometimes be a complex task, in many cases it is relatively straightforward to implement, especially if the expert does not have detailed ideas about possible improvements to the model. In this case, it can be as simple as inflating the variances of some of the uncertain quantities contained in the emulator for f; see, Goldstein, M. and Rougier, J. C. (2009). It should also be stressed that Reification provides a formally consistent approach to linking families of models to reality, in contrast with the Best Input approach.

8.101.3 Additional Comments

A particular example of Reification is provided by Exchangeable Computer Models: see *DiscExchangeableModels* for further discussion of this area.

8.101.4 References

Goldstein, M. and Rougier, J. C. (2009), “Reified Bayesian modelling and inference for physical systems (with Discussion)”, *Journal of Statistical Planning and Inference*, 139, 1221-1239.

8.102 Discussion: Reification Theory

8.102.1 Description and Background

Reification is an approach for coherently linking models to reality, which was introduced in the discussion page *DiscReification*. Here we provide more detail about the various techniques involved in implementing the full Reification process. Readers should be familiar with the basic *emulation* ideas presented in the core threads *ThreadCoreGP* and specifically *ThreadCoreBL*.

8.102.2 Discussion

As covered in *DiscReification*, the Reification approach centres around the idea of linking the current model f to a Reified model f^+. The Reified model incorporates all possible improvements to the model that can be currently imagined. The link to reality :math:y is then given by applying the *Best Input<DefBestInput>* approach (see *DiscBestInput*) to the Reified model only, giving,

$$y = f^+(x^+, w^+) + d^+, \qquad d^+ \perp (f, f^+, x^+, w^+)$$

where w are any extra model parameters that might be introduced due to any of the considered model improvements incorporated by f^+. Hence, this is a more detailed method than the Best Input approach. Here we will describe in more detail how to link a model f to f^+ either directly (a process referred to as Direct Reification) or via an intermediate model f' (a process referred to as Structural Reification). This involves the use of emulators to summarise our beliefs about the behaviour of each of the functions f, f' and f^+. Here f' may correspond to any of the intermediate models introduced in *DiscReification*.

Direct Reification

Direct Reification is a relatively straightforward process where we link the current model f to the Reified model f^+. It should be employed when the expert does not have detailed ideas about specific improvements to the model.

We first represent the model f using an emulator. As described in detail in *ThreadCoreGP* and *ThreadCoreBL*, an emulator is a probabilistic belief specification for a deterministic function. Our emulator for the i might take the form,

$$f_i(x) = \sum_j \beta_{ij}\, g_{ij}(x) + u_i(x)$$

where $B = \{\beta_{ij}\}$ are unknown scalars, g_{ij} are known deterministic functions of the inputs x, and $u_i(x)$ is a weakly stationary stochastic process.

The Reified model might be a function, not just of the current input parameters x , but also of new inputs w which might be included in a future improvement stage. Our simplest emulator for f^+ would then be

$$f_i^+(x,w) = \sum_j \beta_{ij}^+ \, g_{ij}(x) + u_i^+(x) + u_i^+(x,w)$$

where we might model $B^+ = \{\beta_{ij}^+\}$ as $\beta_{ij}^+ = c_{ij}\,\beta_{ij} + \nu_{ij}$ for known c_{ij} and uncertain ν_{ij}, and correlate $u_i(x)$ and $u_i^+(x)$, but leave $u_i^+(x,w)$ uncorrelated with the other random quantities in the emulator. Essentially this step models f^+ by inflating each of the variances in the current emulator for f, and is often relatively simple. We can represent this structure using the following Bayesian Belief Network, where 'child' vertices that are strictly determined by their 'parents' are indicated with dashed lines, where u^+ represents $\{u^+(x), u^+(x,w)\}$ and F represents a collection of model runs:

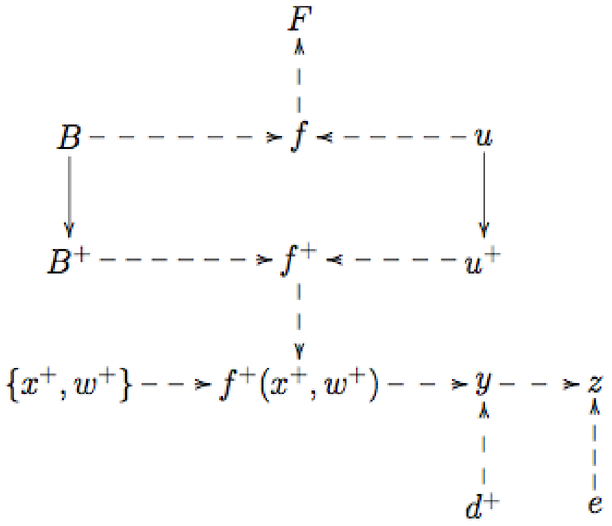


Fig. 38: **Figure 1:** Bayesian belief network for Direct Reification.

Structural Reification

Structural Reification is a process where we link the current model f to an improved model f' , and then to the Reified model f^+. Here f' may correspond to f_{theory} or f'_{theory} which were introduced in *DiscReification*.

Usually, we can think more carefully about the reasons for the model's inadequacy. As we have advocated in the discussion page on expert *assessment* of model discrepancy (*DiscExpertAssessMD*), a useful strategy is to envisage specific improvements to the model, and to consider the possible effects on the model outputs of such improvements. Often we can imagine specific generalisations for $f(x)$ with extra model components and new input parameters v, resulting in an improved model $f'(x,v)$. Suppose the improved model reduces back to the current model for some value of $v = v_0$ i.e. $f'(x,v_0) = f(x)$. We might emulate f' 'on top' of f , using the form:

$$f_i'(x,v) = f_i(x) + \sum_k \gamma_{ik}\, g_{ik}(x,v) + u_i^{(a)}(x,v),$$

where $g_{ik}(x, v_0) = u_i^{(a)}(x, v_0) = 0$. This would give the full emulator for $f_i'(x, v)$ as

$$f_i'(x, v) = \sum_j \beta_{ij}\, g_{ij}(x) + \sum_k \gamma_{ik}\, g_{ik}(x, v) + u_i(x) + u_i^{(a)}(x, v),$$

where for convenience we define $B' = \{\beta_{ij}, \gamma_{ik}\}$ and $u' = u(x) + u^{(a)}(x, v)$. Assessment of the new regression coefficients γ_{ik} and stationary process $u_i^{(a)}(x, v)$ would come from consideration of the specific improvements that f' incorporates. The reified emulator for $f_i^+(x, v, w)$ would then be

$$f_i^+(x, v, w) = \sum_j \beta_{ij}^+\, g_{ij}(x) + \sum_k \gamma_{ik}^+\, g_{ik}(x, v) + u_i^+(x) + u_i^+(x, v) + u_i^+(x, v, w),$$

where we would now carefully apportion the uncertainty between each of the random coefficients in the Reified emulator. Although this is a complex task, we would carry out this procedure when the expert's knowledge about improvements to the model is detailed enough that it warrants inclusion in the analysis. An example of this procedure is given in Goldstein, M. and Rougier, J. C. (2009). We can represent this structure using the following Bayesian Belief Network, with $B^+ = \{\beta_{ij}^+, \gamma_{ik}^+\}$, and $u^+ = u^+(x) + u^+(x, v) + u^+(x, v, w)$:

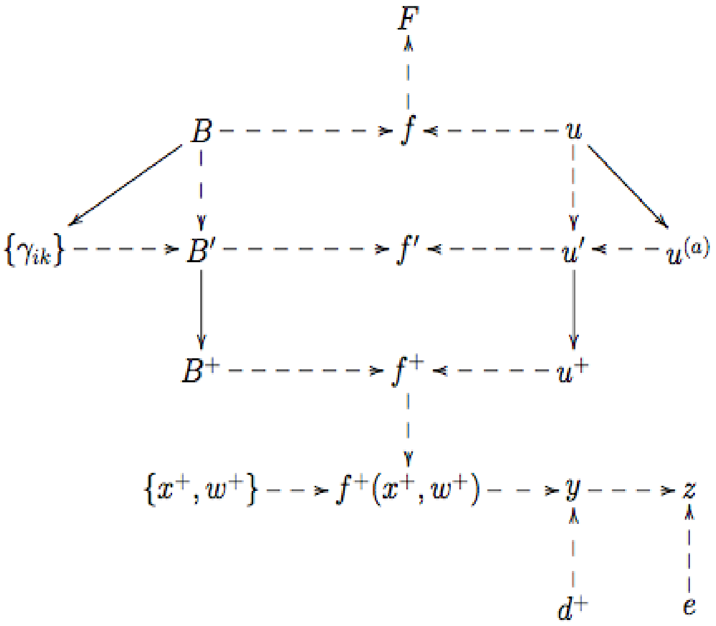


Fig. 39: **Figure 2:** Bayesian belief network corresponding to Structured Reification.

8.102.3 Additional Comments

Once the emulator for f^+ has been specified, the final step is to assess d^+ which links the Reified model to reality. This is often a far simpler process than direct assessment of model discrepancy, as the structured beliefs about the difference between f and reality y have already been accounted for in the link between f, f' and f^+ . Reification also provides a natural framework for incorporating multiple models, by the judgement that each model will be informative about the Reified model, which is then informative about reality y. Exchangeable Computer Models, as described in the discussion page *DiscExchangeableModels*, provide a further example of Reification. More details can be found in Goldstein, M. and Rougier, J. C. (2009).

8.102.4 References

Goldstein, M. and Rougier, J. C. (2009), “Reified Bayesian modelling and inference for physical systems (with Discussion)”, *Journal of Statistical Planning and Inference*, 139, 1221-1239.

8.103 Discussion: Sensitivity measures for decision uncertainty

8.103.1 Description and background

The basic ideas of *sensitivity analysis* (SA) are presented in the topic thread on sensitivity analysis (*ThreadTopicSensitivityAnalysis*). We concentrate in the *MUCM* toolkit on probabilistic SA, but the reasons for this choice and alternative approaches are considered in the discussion page *DiscWhyProbabilisticSA*. *ThreadTopicSensitivityAnalysis* also outlines four general uses for SA; we discuss here the SA measures appropriate to one of those uses - analysing the way that uncertainty concerning the consequences of a decision is induced by uncertainty in its inputs.

Examples of decisions that might be based on simulator output are not difficult to find. Consider a simulator which models the response of the global climate to atmospheric CO_2 levels. The simulator will predict global warming and rising sea levels based on future carbon emissions scenarios, and we can imagine a national policy decision whether to build sea defences to protect a coastal area or city. Uncertainty in the future carbon emissions and climate response to increased CO_2 mean that the consequences of buiding or not building sea defences are uncertain. Another decision based on this simulator might involve setting policy on power station emissions to try to control the nation’s contribution to atmospheric CO_2.

Uncertainty in the inputs is described by a joint probability density function $\omega(x)$, whose specification will be considered as part of the discussion below.

Notation

The following notation and terminology is introduced in *ThreadTopicSensitivityAnalysis*.

In accordance with the standard *toolkit notation*, we denote the simulator by f and its inputs by x. The focus of SA is the relationship between x and the simulator output(s) $f(x)$. Since SA also typically tries to isolate the influences of individual inputs, or groups of inputs, on the output(s), we let x_j be the j-th element of x and will refer to this as the j-th input, for $j = 1, 2, \ldots, p$, where as usual p is the number of inputs. If J is a subset of the indices $\{1, 2, \ldots, p\}$, then x_J will denote the corresponding subset of inputs. For instance, if $J = \{2, 6\}$ then $x_J = x_{\{2,6\}}$ comprises inputs 2 and 6, while x_j is the special case of x_J when $J = \{j\}$. Finally, x_{-j} will denote the whole of the inputs x *except* x_j, and similarly x_{-J} will be the set of all inputs except those in x_J.

8.103.2 Discussion

We consider here how the methods of probabilistic SA can be used for analysing decision uncertainty. All of the SA measures recommended here are defined and discussed in page *DiscDecisionBasedSA*.

Specifying $\omega(x)$

The role of $\omega(x)$ is to represent uncertainty about the simulator inputs x. As such, it should reflect the best available knowledge about the true, but uncertain, values of the inputs. Typically, this distribution will be centred on whatever are the best current estimates of the inputs, with a spread around those estimates that faithfully describes the degree of uncertainty. However, turning this intuitive impression into an appropriate probability distribution is not a simple process.

The basic technique for specifying $\omega(x)$ is known as *elicitation*, which is the methodology whereby expert knowledge/uncertainty is converted into a probability distribution. Some resources for elicitation may be found in the "Additional comments" section below.

Elicitation of a probability distribution for several uncertain quantities is difficult, and it is common to make a simplifying assumption that the various inputs are independent. Formally, this assumption implies that if you were to obtain additional information about some of the inputs (which would generally reduce your uncertainty about those inputs) then your knowledge/uncertainty about the other inputs would not change. Independence is quite a strong assumption but has the benefit that it greatly simplifies the task of elicitation. Instead of having to think about uncertainty concerning all the various inputs together, it is enough to think about the uncertainty regarding each individual input x_j separately. We specify thereby the marginal density function $\omega_j(x_j)$ for each input separately, and the joint density function $\omega(x)$ is just the product of these marginal density functions:

$$\omega(x) = \prod_{j=1}^{p} \omega_j(x_j).$$

Decision under uncertainty

Decisions are hardest to make when their consequences are uncertain. The problem of decision-making in the face of uncertainty is addressed by statistical decision theory. Interest in *simulator* output uncertainty is often driven by the need to make decisions, where the simulator output $f(x)$ is a factor in that decision.

In addition to the joint probability density function $\omega(x)$ which represents uncertainty about the inputs, we need two more components for a formal decision analysis.

1. *Decision set*. The set of available decisions is denoted by $\mathcal{D}$. We will denote an individual decision in $\mathcal{D}$ by d.
2. *Loss function*. The loss function $L(d, x)$ expresses the

   consequences of taking decision d when the true inputs are x.

The interpretation of the loss function is that it represents, on a suitable scale, a penalty for making a poor decision. To make the best decision we need to find the d that minimises the loss, but this depends on x. It is in this sense that uncertainty about (the simulator output and hence about) the inputs x makes the decision difficult. Uncertainty about x leads to uncertainty about the best decision. It is this decision uncertainty that is the focus of decision-based SA.

There is more detailed discussion of the loss function in *DiscDecisionBasedSA*, and examples may be found in the example page *ExamDecisionBasedSA*.

Sensitivity

Consider the effect of uncertainty in a group of inputs x_J; the case of a single input x_j is then included through $J = \{j\}$. As far as the decision problem is concerned, the effect of x_J is shown in the function $M_J(x_J)$. This is the optimal decision expressed as a function of x_J. The optimal decision is the one that we would take if we learnt the true value of x_J (but otherwise learnt nothing about x_{-J}).

If the optimal decision $M_J(x_J)$ were the same for all x_J, then clearly the uncertainty about x_J would be irrelevant, so in some sense the more $M_J(x_J)$ varies with x_J the more influential this group of inputs is. However, whilst it is of interest if the decision changes with x_J, the true measure of importance of this decision uncertainty is whether, by choosing different decisions for different x_J, we expect to make much *better* decisions. That is, how much would we expect the loss to reduce if we were learn the value of x_J? The appropriate measure is the expected value of learning x_J, which is denoted by V_J.

Thus, V_J is our primary SA measure.

Prioritising research

One reason for this kind of SA is to determine whether it would be useful to carry out some research to reduce uncertainty about one or more of the inputs. Decision-based SA is the ideal framework for considering such questions, because we can explicitly value the research. Such research will not usually be able to identify precisely the values of one or more inputs, but V_J represents an upper bound on the value of any research aimed at improving our understanding of x_J.

More precise values can be given to research using the idea of the expected value of sample information (EVSI), which is outlined in *DiscDecisionBasedSA*.

We can compare the value of research directly with what that research would cost. This is particularly easy if the loss function is measured in financial terms, so that V_J (or a more precise EVSI) becomes equivalent to a sum of money. Loss functions for commercial decisions are often framed in monetary terms, but when loss is on some other scale the comparison is less straightforward. Nevertheless, quantifying the effect on decision uncertainty in this way is the best basis for deciding on the cost-effectiveness of research.

8.103.3 Additional comments

The following resources on elicitation will help with the process of specifying $\omega(x)$. The first is a thorough review of the field of elicitation, and provides a wealth of general background information on ideas and methods. The second (SHELF) is a package of documents and simple software that is designed to help those with less experience of elicitation to elicit expert knowledge effectively. SHELF is based on the authors' own experiences and represents current best practice in the field.

O'Hagan, A., Buck, C. E., Daneshkhah, A., Eiser, J. R., Garthwaite, P. H., Jenkinson, D. J., Oakley, J. E. and Rakow, T. (2006). Uncertain Judgements: Eliciting Expert Probabilities. John Wiley and Sons, Chichester. 328pp. ISBN 0-470-02999-4.

SHELF - the Sheffield Elicitation Framework - can be downloaded from http://tonyohagan.co.uk/shelf (*Disclaimer*)

8.104 Discussion: Sensitivity measures for output uncertainty

8.104.1 Description and background

The basic ideas of *sensitivity analysis* (SA) are presented in the topic thread on sensitivity analysis (*ThreadTopicSensitivityAnalysis*). We concentrate in the *MUCM* toolkit on probabilistic SA, but the reasons for this choice and alternative approaches are considered in the discussion page *DiscWhyProbabilisticSA*. *ThreadTopicSensitivityAnalysis* also outlines four general uses for SA; we discuss here the SA measures appropriate to one of those uses - analysing the way that uncertainty concerning a *simulator's* output is induced by uncertainty in its inputs.

Uncertainty in the inputs is described by a joint probability density function $\omega(x)$, whose specification will be considered as part of the discussion below.

Notation

The following notation and terminology is introduced in *ThreadTopicSensitivityAnalysis*.

In accordance with the standard *toolkit notation*, we denote the simulator by f and its inputs by x. The focus of SA is the relationship between x and the simulator output(s) $f(x)$. Since SA also typically tries to isolate the influences of individual inputs, or groups of inputs, on the output(s), we let x_j be the j-th element of x and will refer to this as the j-th input, for $j = 1, 2, \ldots, p$, where as usual p is the number of inputs. If J is a subset of the indices $\{1, 2, \ldots, p\}$, then x_J will denote the corresponding subset of inputs. For instance, if $J = \{2, 6\}$ then $x_J = x_{\{2,6\}}$ comprises inputs

2 and 6, while x_j is the special case of x_J when $J = \{j\}$. Finally, x_{-j} will denote the whole of the inputs x *except* x_j, and similarly x_{-J} will be the set of all inputs except those in x_J.

8.104.2 Discussion

We consider here how the methods of probabilistic SA can be used for analysing output uncertainty. All of the SA measures recommended here are defined and discussed in page *DiscVarianceBasedSA*.

We initially assume that interest focuses on a single simulator output before briefly considering the case of multiple outputs.

Specifying $\omega(x)$

The role of $\omega(x)$ is to represent uncertainty about the simulator inputs x. As such, it should reflect the best available knowledge about the true, but uncertain, values of the inputs. Typically, this distribution will be centred on whatever are the best current estimates of the inputs, with a spread around those estimates that faithfully describes the degree of uncertainty. However, turning this intuitive impression into an appropriate probability distribution is not a simple process.

The basic technique for specifying $\omega(x)$ is known as *elicitation*, which is the methodology whereby expert knowledge/uncertainty is converted into a probability distribution. Some resources for elicitation may be found in the "Additional comments" section below.

Elicitation of a probability distribution for several uncertain quantities is difficult, and it is common to make a simplifying assumption that the various inputs are independent. Formally, this assumption implies that if you were to obtain addtional information about some of the inputs (which would generally reduce your uncertainty about those inputs) then your knowledge/uncertainty about the other inputs would not change. Independence is quite a strong assumption but has the benefit that it greatly simplifies the task of elicitation. Instead of having to think about uncertainty concerning all the various inputs together, it is enough to think about the uncertainty regarding each individual input x_j separately. We specify thereby the marginal density function $\omega_j(x_j)$ for each input separately, and the joint density function $\omega(x)$ is just the product of these marginal density functions:

$$\omega(x) = \prod_{j=1}^{p} \omega_j(x_j).$$

Independence also simplifies the interpretation of some of the SA measures described in *DiscVarianceBasedSA*.

Uncertainty analysis

Uncertainty about x induces uncertainty in the simulator output $f(x)$. The task of measuring and describing that uncertainty is known as *uncertainty analysis* (UA). Formally, we regard the uncertain inputs as a random variable X (conventionally, random variables are denoted by capital letters in Statistics). Then the output $f(X)$ is also a random variable and has a probability distribution known as the uncertainty distribution. Some of the most widely used measures of output uncertainty in UA are as follows.

- The distribution function $F(c) = Pr(f(X) \le c)$.
- The uncertainty mean $M = \mathrm{E}[f(X)]$.
- The uncertainty variance $V = \mathrm{Var}[f(X)]$.
- The exceedance probability $\bar{F}(c) = 1 - F(c)$, that $f(X)$ exceeds some threshhold c.

Sensitivity

The goal of SA, as opposed to UA, is to analyse the output uncertainty so as to understand which uncertain inputs are most responsible for the output uncertainty. If uncertainty in a given input x_j is accountable for a large part of the output uncertainty, then the output is said to be very sensitive to x_j. Therefore, SA explores the relative sensitivities of the inputs, both individually and in groups.

Output uncertainty is primarily summarised by the variance V. As defined in *DiscVarianceBasedSA*, the proportion of this overall variance that can be attributed to a group of inputs x_J is given by the sensitivity index S_J or by the total sensitivity index T_J.

Formally, S_J is the expected amount by which uncertainty would be reduced if we were to learn the true values of the inputs in x_J. For instance, if $S_J = 0.25$ then learning the true value of x_J would reduce output uncertainty by 25%.

On the other hand, T_J is the expected proportion of uncertainty remaining if we were to learn the true values of all the *other* inputs, i.e. x_{-J}. As explained in *DiscVarianceBasedSA*, when inputs are independent T_J will be larger than S_J by an amount indicating the magnitude of interactions between inputs in x_J and other inputs outside the group.

If there is an interaction between inputs x_j and $x_{j'}$ then (again assuming independence) the sensitivity index $S_{\{j,j'\}}$ for the two inputs together is greater than the sum of their individual sensitivity indices S_j and $S_{j'}$. So interactions can be important in identifying which groups of inputs have the most influence on the output.

Prioritising research

One reason for this kind of SA is to determine whether it would be useful to carry out some research to reduce uncertainty about one or more of the inputs. In general, there would be little value in conducting such research to learn about the true value of an input whose sensitivity index is very small. An input (or input group) with a high sensitivity index is more likely to be a priority for research effort.

However, a more careful consideration of research priorities would involve other factors. First of these would be cost. An input may have a high sensitivity index but still might not be a priority for research if the cost of investigating it would be very high. Another factor is the purpose for which the research is envisaged. The primary objective may be more complex than simply reducing uncertainty about $f(X)$. The reason why we are interested in $f(X)$ in the first place is likely to be as an input to some decision problem, and the importance of input uncertainty is then not simply that it causes output uncertainty but that it causes decision uncertainty, i.e. uncertainty about the best decision. This takes into the realm of decision-based SA; see *DiscDecisionBasedSA*.

Exceedances

Although overall uncertainty, as measured by V, is generally the most important basis for determining sensitivity, interest may sometimes focus on other aspects of the uncertainty distribution. If there is a decision problem, for instance, even if we do not wish to pursue the methods of decision-based SA, the decision context may suggest some function of the output $f(X)$ that is of more interest than the output itself. Then SA based on the variance of that function may be more useful. We present a simple example here.

Suppose that interest focuses on whether $f(X)$ exceeds some threshhold c. Instead of $f(x)$ we consider as our output $f_c(x)$, which takes the value $f_c(x) = 1$ if $f(X) > c$ and otherwise $f_c(x) = 0$. Now the uncertainty mean M is just the exceedance probability $M = \bar{F}(c)$ and the uncertainty variance can be shown to be $V = \bar{F}(c)\{1 - \bar{F}(c)\}$.

The mean effect of inputs x_J becomes

$$M_J(x_J) = Pr(f(X) > c \,|\, x_J)$$

and the sensitivity variance V_J is the variance of this mean effect with respect to the distribution of x_J. The corresponding sensitivity index $S_J = V_J/V$ then measures the extent to which x_J influences the probability of exceedance.

Multiple outputs

When the simulator produces multiple outputs, then we may be interested in uncertainty about all of these outputs. Although *DiscVarianceBasedSA* describes how then we can generalise the sensitivity variance V_J to a matrix, there is generally little extra value to be gained from looking at sensitivity of multiple outputs in this way. It is usually adequate to identify the inputs that most influence each of the outputs separately. However, this will typically lead to different groups of inputs being most important for different inputs, and it is no longer clear which ones are candidates for research prioritisation. In practice, the solution is again to think about the decision context and to use the methods of decision-based SA.

8.104.3 Additional comments

Transformation of the output may make for simpler SA. If, for instance, $f(x)$ must be positive but can vary through two or more orders of magnitude (a factor of 100 or more) then working with its logarithm, $\log f(x)$, is worth considering. There may be fewer important inputs and fewer interactions on the logarithmic scale.

The following resources on elicitation will help with the process of specifying $\omega(x)$. The first is a thorough review of the field of elicitation, and provides a wealth of general background information on ideas and methods. The second (SHELF) is a package of documents and simple software that is designed to help those with less experience of elicitation to elicit expert knowledge effectively. SHELF is based on the authors' own experiences and represents current best practice in the field.

O'Hagan, A., Buck, C. E., Daneshkhah, A., Eiser, J. R., Garthwaite, P. H., Jenkinson, D. J., Oakley, J. E. and Rakow, T. (2006). Uncertain Judgements: Eliciting Expert Probabilities. John Wiley and Sons, Chichester. 328pp. ISBN 0-470-02999-4.

SHELF - the Sheffield Elicitation Framework - can be downloaded from http://tonyohagan.co.uk/shelf (*Disclaimer*)

8.105 Discussion: Sensitivity analysis measures for simplification

8.105.1 Description and background

The basic ideas of *sensitivity analysis* (SA) are presented in the topic thread on sensitivity analysis (*ThreadTopicSensitivityAnalysis*). We concentrate in the *MUCM* toolkit on probabilistic SA, but the reasons for this choice and alternative approaches are considered in the discussion page *DiscWhyProbabilisticSA*. *ThreadTopicSensitivityAnalysis* also outlines four general uses for SA; we discuss here the SA measures appropriate to two of those uses - understanding the way that a *simulator's* output responds to changes in its inputs, and identifying inputs with little effect on the output with a view to simplifying either the simulator or a corresponding *emulator*. We refer to both of these uses as simplification because understanding is generally best achieved by looking for simple descriptions.

In probabilistic SA, we assign a function $\omega(x)$ that is formally treated as a probability density function for the inputs. The interpretation and specification of $\omega(x)$ will be considered as part of the discussion below.

Notation

The following notation and terminology is introduced in *ThreadTopicSensitivityAnalysis*.

In accordance with the standard *toolkit notation*, we denote the simulator by f and its inputs by x. The focus of SA is the relationship between x and the simulator output(s) $f(x)$. Since SA also typically tries to isolate the influences of individual inputs, or groups of inputs, on the output(s), we let x_j be the j-th element of x and will refer to this as the j-th input, for $j = 1, 2, \ldots, p$, where as usual p is the number of inputs. If J is a subset of the indices $\{1, 2, \ldots, p\}$, then x_J will denote the corresponding subset of inputs. For instance, if $J = \{2, 6\}$ then $x_J = x_{\{2,6\}}$ comprises inputs

2 and 6, while x_j is the special case of x_J when $J = \{j\}$. Finally, x_{-j} will denote the whole of the inputs x *except* x_j, and similarly x_{-J} will be the set of all inputs except those in x_J.

8.105.2 Discussion

We consider here how the methods of probabilistic SA can be used for understanding and dimension reduction. We also consider the related topic of validation/criticism of a simulator. All of the SA measures recommended here are defined and discussed in page *DiscVarianceBasedSA*.

We assume that interest focuses on a single simulator output. Understanding of several outputs is best done by exploring each output separately. However, some remarks on dimension reduction for multiple outputs may be found in the "Additional comments" section.

Specifying $\omega(x)$

Although $\omega(x)$ is treated formally as a probability density function in probabilistic SA, for the purposes of simplification it is often simply a weight function specifying the relative importance of different x values. Indeed, it is often a uniform density giving equal importance for all points in a subset of the possible x space.

Weight function

Note that we generally wish to study the behaviour of the simulator over a subset of the whole space of possible inputs, because we are interested in applications of that simulator to model a particular instance of the real-world processes being simulated. For example, a simulator of rainfall runoff through a river catchment has inputs which range over values such that it can predict flows for a wide variety of real river catchments, but in a given situation we are interested in its behaviour for a particular catchment. For this particular instance, the inputs will have a narrower range to represent the likely values in that catchment.

Suppose first that we wish to weight all points in the subset equally, and so use a uniform weight function. If the range of values of input x_j is specified as $a_j \le x_j \le b_j$, for $j = 1, 2, \ldots, p$, then the area of the subset of interest is

$$A = \prod_{j=1}^{p}(b_j - a_j),$$

and the uniform density for values of x in this subset is

$$\omega(x) = 1/A,$$

with $\omega(x) = 0$ for x outside that subset. [Since $\omega(x)$ is treated as a probability density function, the total weight over the subset must be 1, hence the weight at each point is $1/A$.] Note that in this case the inputs are independent, which leads to some simplification of the sensitivity measures presented in *DiscVarianceBasedSA*.

If we are more interested in some parts of the input space than in others, then we could use a non-uniform weight function.

Probability density

We can also consider situations where it would be appropriate to specify $\omega(x)$ as a genuine probability density function. One is where there is uncertainty about the proper input values to use, and our wish for understanding or dimension reduction is in the context of that uncertainty. In this case, we may also want to carry out SA for analysing output uncertainty or decision uncertainty, but understanding and/or dimension reduction are useful preliminary explorations.

The appropriate choice of $\omega(x)$ in this case is the probability distribution that represents the uncertainty about x; see the discussion page on sensitivity measures for output uncertainty (*DiscSensitivityAndOutputUncertainty*).

Another case can be identified by considering again the example of a simulator of rainfall runoff in a river catchment. For applications in a given catchment, over a period of time, inputs defining the rainfall incidence and amounts of water already in the catchment will vary. A probability distribution might be chosen to represent the relative prevalence of different conditions in the catchment.

Dimensionality reduction

Simulators often have a large number of inputs. Whilst these are all considered relevant and are all expected to have some impact on the outputs, many will have only minor effects. This is particularly true when we are interested in using the simulator for a specific application where the inputs have restricted ranges. The response of the simulator outputs over the input subspace of interest may be dominated by only a small number of inputs. The goal of dimensionality reduction is to separate the important inputs from those to which the output is relatively insensitive.

With a large number of inputs, it becomes impractical to consider varying all the inputs together to perform a full SA. In practice, various simplified *screening* procedures are generally used to reduce the set of inputs to a more manageable number. A discussion of screening methods is available at the topic thread on screening (*ThreadTopicScreening*).

Formal SA techniques are typically use to explore more carefully the inputs which cannot be eliminated by simple screening tools, with a view to finding the small number of most influential inputs. It is in this final exploratory and confirmatory phase of dimension reduction that SA methods are most useful.

The SA measure that is best for identifying an input that has little effect on the simulator output is its total sensitivity index T_j. This is the proportion of uncertainty that would remain if all the remaining inputs x_{-j} were known. In the case of independent inputs, any x_j for which T_j is less than, say, 0.01 (or 1%) could be considered to have only a very small effect on the output. If inputs are not independent, small T_j does not necessarily imply that x_j has little effect, and it is important to check also that its sensitivity index S_j is small.

Understanding

Understanding is a less well defined goal than dimension reduction, and a variety of measures may be useful. Dimension reduction is a good start, since much simplification is achieved by reducing the number of inputs to be considered because their effects are appreciable.

Having identified the important inputs, another very useful step is to split these into groups with only minimal interaction between groups. Then the response of the output can be considered as a sum of effects due to the separate groups. A group of inputs x_J has negligible interaction with the remaining inputs x_{-J} if their group sensitivity measure S_J is close to their total sensitivity measure T_J.

When the important inputs have been subdivided in this way, understanding is achieved by looking at the behaviour of the output in response to each group separately. For a group x_J, the most useful SA measure now is its mean effect $M_J(x_J)$. If the group comprises just a single input, then its mean effect can simply be plotted to provide a visual impression of how that input affects the output. This can be supplemented by looking at regression components. For instance, if the effect of this input is nearly linear then its linear variance measure V_j^L will be close to its variance V_j, and the slope of the linear regression line will provide a simple description of how the output responds to this input.

If a group is not single input, and it cannot be split further, then there are appreciable interactions between inputs within the group. We can still examine the mean effect $M_j(x_j)$ of each individual input in the group, but we also need to consider the joint effect. Plotting the two-input mean effect $M_{\{j,j'\}}(x_{\{j,j'\}})$ of a pair of inputs as a contour plot can give a good visual impression of their joint effect. Similarly, we could view a contour plot of their interaction effect $I_{\{j,j'\}}(x_{\{j,j'\}})$, but for a group with more than two inputs it is difficult to get understanding beyond pairwise effects. Careful exploration of regression terms (analogously to the conventional statistical method of stepwise regression) may succeed in identifying a small number of dominant terms - some discussion of this may be provided in a future version of the toolkit.

Criticism/validation

A simulator is a model of some real-world process. A wish to understand or simplify a simulator may be motivated by wanting to understand the real-world process. It is obviously unwise to use the simulator as a surrogate for reality in this way unless it is known that the simulator is a good representation of reality. Techniques for modelling and validating the relationship between a simulator and reality will be introduced in a later version of the toolkit.

However, there is a useful, almost opposite use for understanding a simulator. We will often have some clear qualitative understanding or intuition about how reality behaves, and by examining the behaviour of the simulator we can check whether it conforms to such understanding/intuition. If, for instance, we expect increasing x_1 to cause the real-world value $y(x)$ to increase, then we will be worried if we find that $f(x)$ generally decreases when x_1 increases. Similarly, we can check whether the most active inputs, the presence or absence of interactions and the nature of nonlinearity in the simulator agrees with how we expect reality to behave.

If there is a mismatch between behaviour of the simulator, as revealed by SA techniques, and our beliefs about reality then this suggests that one of them is faulty. Either reality does not actually behave as we think it should, or else the simulator does not capture reality adequately. We cannot expect a simulation to be a perfectly accurate representation of reality, but if we find that its qualitative behaviour is wrong then this often suggests the kind of modification that might be made to improve the simulator.

8.105.3 Additional comments

Transformation of the output may make it easier to find simplification. If, for instance, $f(x)$ must be positive but can vary through two or more orders of magnitude (a factor of 100 or more) then working with its logarithm, $\log f(x)$, is worth considering. There may be fewer important inputs, fewer interactions and less nonlinearity on the logarithmic scale.

In the case of multiple outputs, if dimension reduction is applied to each output separately this will typically lead to a different set of most active inputs being retained for different outputs. If there is a wish to simplify the simulator by eliminating the same set of inputs for all outputs, this can again be done by considering the total sensitivity index :math:T_j, but now this is a matrix (see *DiscVarianceBasedSA<DiscVarianceBasedSA>*). An input should be considered to have a suitably small effect if *all* the elements of T_j are small.

8.106 Discussion: Sobol sequence

8.106.1 Description and background

A Sobol sequence is a sequence of points in a design space which has the general property of being *space filling*. The procedure page *ProcSobolSequence* provides a detailed algorithm for constructing Sobol sequences based on Sobol's original description, and a worked example. Sobol's construction is based on bit-wise XOR operations between special generators called direction numbers. The first impression is that the construction of a Sobol sequence is a complex task. However, by constructiong a few Sobol numbers by hand, using the procedure in *ProcSobolSequence*, it is easy to understand the underlying structure.

8.106.2 Discussion

For users of the R programming language, we suggest the function `runif.sobol(n,d)` from the package fOptions in the R repository.

A well known concern about Sobol sequences is the correlation between points in high dimensional sequences. A solution to this is by scrambling the sequence. The R package above described allows for the sequence to be scrambled, if desired. In this case, either of the two functions described above can be called with the arguments

```
(n,d,scrambling,seed,init)
```

The argument scrambling takes an integer which is 0 if no scrambling is to be used, 1 for Owen type of scrambling, 2 for Faure-Tezuka scrambling and 3 for a combination of both. The argument seed is the integer seed to trigger the scrambling process and init is a logical flag which allows the Sobol sequence to restart if set to `TRUE`.

8.107 Discussion: Structured Forms for Model Discrepancy

8.107.1 Description and background

The basic ideas of *model discrepancy* are presented in the variant thread on linking models to reality (*ThreadVariantModelDiscrepancy*). There, the key equations are $y = f(x^+) + d$ and $z = y + \epsilon$, where y represents system values, f is the *simultaor*, x^+ is the *best input* and z are observations on y with measurement error ϵ (*DiscObservations*). We expand that notation here to allow explicit indexing of these quantities by other inputs; for example, space-time co-ordinates. Possible situations where d has a highly structured form due to physical structures present in the model are discussed.

8.107.2 Discussion

In accordance with the standard toolkit notation, we denote the model simulator by f and its inputs by x. We extend the notation used in *ThreadVariantModelDiscrepancy* to include generalised ‘location’ inputs u; for example, u might index space-time, such as latitude, longitude and date, and $y(u)$ might denote the real average temperature of the earth’s surface at that latitude and longitude over a particular day. We write $f(u, x)$ and $y(u)$ to denote the simulator output for inputs x at location u and the system value at location u, respectively. The fundamental relationship linking model to reality given in *ThreadVariantModelDiscrepancy* now becomes

$$y(u) = f(u, x^+) + d(u)$$

where $d(u)$ is the model discrepancy between the real system value $y(u)$ at u and the simulator value $f(u, x^+)$ for the best input x^+ at u: for detailed discussions about x^+ and $d(u)$ see *DiscBestInput* and *DiscWhyModelDiscrepancy*. As there may be replicate observations of the real system at a location u, we introduce notation to cover this case. Denote by $z_r(u)$ the r-th replicate observation of the system value $y(u)$ at location u. As in *ThreadVariantModelDiscrepancy* and *DiscObservations*, we assume that the $z_r(u)$ and the real system value $y(u)$ are additively related through the observation equation

$$z_r(u) = y(u) + \epsilon_r(u)$$

where $\epsilon_r(u)$ is the measurement error of the r-th replicate observation on $y(u)$. We write $\epsilon(u)$ when there is no replication. Combining these equations, we obtain the following relationship between system observations and simulator output

$$z_r(u) = f(u, x^+) + d(u) + \epsilon_r(u)$$

Statistical assumptions

1. The components of the collection $f(u, \cdot)$, x^+, $d(u)$, $\epsilon_r(u)$ are assumed to be independent random quantities for each u in the collection U of u-values considered, or just uncorrelated in the Bayes linear case (*ThreadCoreBL*).
2. The distribution of $f(u, \cdot)$, x^+, $d(u)$, $\epsilon_r(u)$ over all finite collections of u-values in U needs to be specified in any particular application.

3. The model discrepancy term $d(\cdot)$ is regarded as a random process over the collection U. Either the distribution of $d(\cdot)$ is specified over all finite subsets of U, such as a *Gaussian process* for a full Bayes analysis (*ThreadCoreGP*), or just the expectation and covariance structure of $d(\cdot)$ are specified over all finite subsets of U as is required for a Bayes linear analysis (*ThreadCoreBL*). A separable covariance structure for $d(\cdot)$, similar to that defined in *DefSeparable*, might be a convenient simple choice when, for example, u indexes space-time.

4. As the system is observed at only a finite number of locations $u_1, \ldots, u_k$ (say), unlike $d(\cdot)$, we do not regard $\epsilon_r(\cdot)$ as a random process over U. In most applications, the measurement errors $\epsilon_{r_i}(u_i)$ are assumed to be independent and identically distributed random quantities with zero mean and variance Σ_ϵ. However, measurement errors may be correlated across the locations $u_1, \ldots, u_k$, which might be the case, for example, when u indexes time. On the other hand, the replicate observations within each of locations $u_1, \ldots, u_k$ are almost always assumed to be independent and identically distributed; see *DiscObservations*.

8.107.3 Additional comments and Examples

Kennedy, M. C. and O'Hagan, A. (2001) generalise the combined equation to

$$z_r(u) = \rho f(u, x^+) + d(u) + \epsilon_r(u)$$

where ρ is a parameter to be specified or estimated.

Rainfall runoff

Iorgulescu, I., Beven, K. J., and Musy, A. (2005) consider a rainfall runoff model which simulates consecutive hourly measurements of water discharge D and Calcium C and Silica S concentrations for a particular catchment area. Here, $u = t$ is hour t and $y(t) = (D(t), C(t), S(t))$ is the amount of water discharged into streams and the Calcium and Silica concentrations at hour t. There were 839 hourly values of t. While the authors did not consider model discrepancy explicitly, a simple choice would be to specify the covariance $\text{Cov}[d(t_k), d(t_\ell)]$ between $d(t_k)$ and $d(t_\ell)$ to be of the form

$$\sigma^2 \exp\left(-\theta(t_k - t_\ell)^2\right)$$

A more realistic simple choice might be the covariance structure derived from the autoregressive scheme $d(t_{k+1}) = \rho d(t_k) + \eta_k$, where η_k is a white noise process with variance σ^2; the parameters ρ and σ^2 are to be specified or estimated. Such a covariance structure would be particularly appropriate when forecasting future runoff.

The input vector x has eighteen components. An informal *assessment* of model discrepancy for this runoff model is given in Goldstein, M., Seheult, A. and Vernon, I. (2010): see also *DiscInformalAssessMD*.

Galaxy formation

Goldstein, M. and Vernon, I. (2009) consider the galaxy formation model 'Galform' which simulates two outputs, the b_j and K band luminosity functions. The b_j band gives numbers of young galaxies s per unit volume of different luminosities, while the K band describes the number of old galaxies l. The authors consider 11 representative outputs, 6 from the b_j band and 5 from the K band. Here, $u = (A, \lambda)$ is age A and luminosity λ and $y(A, \lambda)$ is count of age A galaxies of luminosity λ per unit volume of space. The authors carried out a careful elicitation process with the cosmologists for $\log y$ and specified a covariance $\text{Cov}[d(A_i, \lambda_l), d(A_j, \lambda_k)]$ between $d(A_i, \lambda_l)$ and $d(A_j, \lambda_k)$ of the form

$$a \begin{bmatrix} 1 & b & .. & c & .. & c \\ b & 1 & .. & c & . & c \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ c & .. & c & 1 & b & .. \\ c & .. & c & b & 1 & .. \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{bmatrix}$$

for specified values of the overall variance a, the correlation within bands b and the correlation between bands c. The input vector x has eight components.

Hydrocarbon reservoir

Craig, P. S., Goldstein, M., Rougier, J. C., and Seheult, A. H. (2001) consider a hydrocarbon reservoir model which simulates bottom-hole pressures of different wells through time. Here, $u = (w, t)$ is the pair well w and time t and $y(w, t)$ is the bottom-hole pressure for well w at time t. There were 34 combinations of w and t considered. The authors specify a non-separable covariance $\mathrm{Cov}[d(w_i, t_k), d(w_j, t_\ell)]$ between $d(w_i, t_k)$ and $d(w_j, t_\ell)$ of the form

$$\sigma_1^2 \exp\left(-\theta_1(t_k - t_\ell)^2\right) + \sigma_2^2 \exp\left(-\theta_2(t_k - t_\ell)^2\right) I_{w_i = w_j}$$

where I_P denotes the indicator function of the proposition P. The input vector x has four active components: see *DefActiveInput*.

Spot welding

Higdon, D., Kennedy, M., Cavendish, J. C., Cafeo, J. A., and Ryne, R. D. (2004) consider a model for spot welding which simulates spot weld nugget diameters for different combinations of load, and current applied to two metal sheets, and gauge is the thickness of the two sheets. Here, $u = (l, c, g)$ is the triple load l, current c and gauge g and $y(l, c, g)$ is the weld diameter when load l and current c are applied to sheets of gauge g at the $12 = 2 \times 3 \times 2$ combinations. Moreover, there is system replication of 10 observations for each of the 12 system combinations. The authors specify a Gaussian process for the model discrepancy d over (l, c, g) with a separable covariance structure. The input vector x has one component.

8.107.4 References

Craig, P. S., Goldstein, M., Rougier, J. C., and Seheult, A. H. (2001), “Bayesian forecasting for complex systems using computer simulators”, *Journal of the American Statistical Association*, 96, 717-729.

Goldstein, M. and Vernon, I. (2009), “Bayes linear analysis of imprecision in computer models, with application to understanding the Universe”, in 6th International Symposium on Imprecise Probability: Theories and Applications.

Goldstein, M., Seheult, A. and Vernon, I. (2010), “Assessing Model Adequacy”, MUCM Technical Report 10/04.

Higdon, D., Kennedy, M., Cavendish, J. C., Cafeo, J. A., and Ryne, R. D. (2004), “Combining field data and computer simulations for calibration and prediction”, *SIAM Journal on Scientific Computing*, 26, 448-466.

Iorgulescu, I., Beven, K. J., and Musy, A. (2005), “Data-based modelling of runoff and chemical tracer concentrations in the Haute-Mentue research catchment (Switzerland)”, *Hydrological Processes*, 19, 2557-2573.

Kennedy, M. C. and O’Hagan, A. (2001), “Bayesian calibration of computer models”, *Journal of the Royal Statistical Society, Series B*, 63, 425-464.

8.108 Discussion: Use of a structured mean function

8.108.1 Overview

There are differing views on an appropriate degree of complexity for an emulator mean function, $m_\beta(x)$, within the computer model community. Both using very simple mean functions, such as a constant, and using more structured linear forms have support. The purpose of this page is to outline the advantages of including a structured mean function into an emulator where appropriate.

8.108.2 Discussion

When working with expensive computer models, the number of available evaluations of the accurate computer model are heavily restricted and so we will obtain poor coverage of the input space for a fixed design size. In such settings, using a simple constant-only emulator mean function will result in emulators which are only locally informative in the vicinity of points which have been evaluated and will revert to their constant prior mean value once we move into regions of input space which are far from observed model runs. Conversely, a regression model contains global terms which are informative across the whole input space and so does not suffer from these problems of sparse coverage, even when far from observed simulator runs

In cases where detailed prior beliefs exist about the qualitative global behaviour of the function, then a logical approach is to construct an appropriate mean function which captures this. Global aspects of simulator behaviour such as monotonicity or particular structural forms are hard to capture from limited simulator evaluations without explicit modelling in the prior mean function. By exposing potential global effects via our prior beliefs, this allows us to learn about these effects and directly gain specific quantitative insight into the simulator's global behaviour, something which would require extensive sensitivity analysis with a constant-only mean function. Additionally, by considering how the global model effects may change over time and space (or more generally across different outputs) the structured mean functions provide a natural route into emulation problems with multivariate (spatio-temporal) output.

If we can explain much of the simulator's variation by using a structured mean function, then this has the added effect of removing many of the simulator's global effects from the stochastic residual process. Indeed, Steinberg & Bursztyn (2004) demonstrated that a stochastic process with a Gaussian correlation function incorporates low-order polynomials as leading eigenfunctions. Thus by explicitly modelling simple global behaviours of the simulator via a structured mean function we remove the global effects which were hidden in the "black box" of the stochastic process, and thus reduce the importance of the stochastic residual process. By diminishing the influence of the stochastic process, many calculations involving the emulator can be substantially simplified – in particular, the construction of additional designs and the visualisation of the emulator when the simulator output is high-dimensional. Furthermore, capturing much of the simulator variation in the mean function reduces the importance of challenging and intensive tasks such as the estimation of the hyperparameters δ of the correlation function.

8.108.3 Overfitting

The primary potential disadvantage to using a structured mean function is that there is the potential for overfitting of the simulator output. If the mean function is overly complex relative to the number of available model evaluations, such as having specified a large number of basis functions or as the result of empirical construction, then our emulator mean function may begin to fit the small-scale behaviours of the residual process. The consequences of overfitting are chiefly poor predictive performance in terms of both interpolation and extrapolation, and over-confidence in our adjusted variances. For obvious reasons, simple mean functions do not encounter this problem however they also do not have the benefits of adequately expressing the global model effects. The risk of overfitting can be best controlled via appropriate diagnostics and validation and the use of an appropriate parsimonious mean function.

8.108.4 References

- Steinberg, D. M., and Bursztyn, D. (2004) "Data Analytic Tools for Understanding Random Field Regression Models," *Technometrics*, **46**:4, 411-420

8.109 Discussion: Sensitivity analysis in the toolkit

8.109.1 Description and background

This page is part of the topic thread on *sensitivity analysis* (SA). The basic ideas of SA are introduced in the topic thread on sensitivity analysis (*ThreadTopicSensitivityAnalysis*), where the various uses of SA are outlined. In the *MUCM* toolkit, the favoured method of SA is probabilistic SA, see the discussion page *DiscWhyProbabilisticSA*. Measures of sensitivity for two particular forms of probabilistic SA are developed in the discussion pages on Variance based SA (*DiscVarianceBasedSA*) and decision based SA (*DiscDecisionBasedSA*). Practical issues concerning the usage of these methods are discussed in the sensitivity analysis measures for simplification page (*DiscSensitivityAndSimplification*), the sensitivity measures for output uncertainty page (*DiscSensitivityAndOutputUncertainty*) and the sensitivity measures for decision uncertainty page (*DiscSensitivityAndDecision*). Here we consider how SA is dealt with in the toolkit, and provide links to relevant pages within the main toolkit threads.

The MUCM toolkit is primarily concerned with methods to understand and manage uncertainty in the outputs of a *simulator* through building an *emulator*. The key advantage of this approach is computational. Simulators are often very large computer programs requiring substantial computer power and time to run. Consequently, the number of times that the simulator can be run is typically small, and this makes it difficult to carry out various kinds of tasks that we might wish to perform using the simulator. The MUCM approach consists basically of first using a relatively small number of runs to build the emulator, and then using the emulator (rather than the simulator itself) to carry out the desired task.

SA is a very good example of a task that can be tackled more efficiently using emulators. The various SA measures presented in *DiscVarianceBasedSA* or *DiscDecisionBasedSA* all involve integration of the simulator output with respect to uncertainty in one or more of the inputs. In order to compute such integrals exactly, we must effectively run the simulator at every possible value of the uncertain inputs, which (in theory at least) requires an infinite number of runs. Any attempt to compute such an integral in practice is therefore subject to computational error. A widely used method is Monte Carlo, in which a random sample of values are drawn for the uncertain inputs, and the simulator is run for each of the sampled input sets. To make the sampling error in the resulting estimates of SA measures appropriately small will typically require thousands of simulator runs (and fresh samples are usually required to compute each SA measure of interest). Using emulators, the number of runs required for comparable levels of computational accuracy may be orders of magnitude smaller.

The discussion here is in two parts. First we consider how the additional uncertainty due to emulation (known as *code uncertainty*) interacts with the measures of sensitivity. The second part points to where specific methods for SA computations are presented in the toolkit's main threads (i.e. the core, variant and generic threads; see *MetaToolkitStructure*).

8.109.2 Discussion

Code uncertainty

Notation

Computation of any SA measure, such as the sensitivity index V_j for input x_j, using emulator techniques is subject to code uncertainty. The emulator is an expression of our knowledge or beliefs about a simulator's output $f(x)$ based on a *training sample* of simulator runs. In the fully *Bayesian* form of emulator based on a *Gaussian process* (GP), the emulator is a posterior distribution for $f(x)$. A *Bayes linear* (BL) emulator provides instead the adjusted mean and variance for $f(x)$ based on the training sample. The posterior distribution or adjusted mean and variance represent uncertainty about $f(x)$ due to emulation, called code uncertainty.

Consequently, any computation of SA measures is subject also to code uncertainty; expressed as a posterior distribution in the case of GP emulation or adjusted mean and variance for BL emulation. The posterior mean or adjusted mean, denoted by E^*, is the computed value or estimate of the measure, and the posterior variance or adjusted variance, denoted by Var^*, characterises the degree of code uncertainty about its value.

For example, $\mathrm{E}^*[V_j]$ is the computed value or estimate of the sensitivity index V_j, with code uncertainty quantified by $\mathrm{Var}^*[V_j]$.

Code uncertainty in variance-based SA

In both probabilistic SA and *uncertainty analysis* (UA) the focus is on uncertainty that is induced by uncertainty about simulator inputs, and the question arises as to whether code uncertainty should be combined in some way with input uncertainty. If we first consider UA, then as described in *DiscSensitivityAndOutputUncertainty* the uncertainty variance $V = \mathrm{Var}[f(X)]$ expresses overall uncertainty about the simulator output $f(X)$ when there is uncertainty about the inputs (in which case we conventionally use the capital letter X to rerpresent a random variable). Similarly, $M = \mathrm{E}[f(X)]$ is generally thought of as the best estimate of $f(X)$ in the presence of input uncertainty. But uncertainty about the simulator output is not confined to uncertainty about X; we are also uncertain about the simulator itself, $f(\cdot)$, and this uncertainty is code uncertainty.

In the presence of both input and code uncertainty, we would estimate $f(x)$ by

$$M^* = \mathrm{E}^*[M],$$

with overall uncertainty comprising

$$V^* = \mathrm{E}^*[V] + \mathrm{Var}^*[M].$$

So in addition to the posterior estimated or computed value of V we also have code uncertainty about M.

In variance based SA, set out in *DiscVarianceBasedSA*, we consider what part of the overall output uncertainty V is attributable to a single input x_j or a group of inputs x_J. Should code uncertainty also be a part of these computations? In particular, should variance based SA analyse how much of V^* (rather than V) is attributable to x_j or x_J?

The answer basically is that learning about the uncertain inputs does not affect code uncertainty. No matter how much and in what way we reduce uncertainty about X, the code uncertainty $\mathrm{Var}^*[M]$ will always be a component of the overall uncertainty about $f(x)$. The computed sensitivity variances $\mathrm{E}^*[V_J]$ or interaction variances $\mathrm{E}^*[V^I_J]$ quantify reductions in the uncertainty of $f(x)$, whether we think of this as being just the computed input uncertainty variance $\mathrm{E}^*[V]$ or the overall variance V^*.

In the case of independent inputs, the computed main effect and interaction variances, $\mathrm{E}^*[V_j]$ and $\mathrm{E}^*[V^I_J]$, provide a partition of $\mathrm{E}^*[V]$ in exactly the way described in the discussion page on the theory of variance based SA (*DiscVarianceBasedSATheory*), while V^* is partitioned by these terms plus the code uncertainty $\mathrm{Var}^*[M]$. We can define a sensitivity index for x_J as $\mathrm{E}^*[V_J]/\mathrm{E}^*[V]$ or allowing for code uncertainty as $\mathrm{E}^*[V_J]/V^*$. In the latter case, code uncertainty has its own index $\mathrm{Var}^*[M]/V^*$.

Code uncertainty in decision-based SA

The situation in decision-based SA is potentially more complex because code uncertainty is not just a fixed addition to the computed values of the measures presented in *DiscDecisionBasedSA*. We can contrast, on the one hand, decision making in the presence of code uncertainty with, on the other hand, code uncertainty about a decision.

If we simply consider the code uncertainty as part of the overall uncertainty about the simulator outputs, along with that induced by input uncertainty, then we define the expected utility to be the expectation with respect to both code uncertainty and input uncertainty, and so define

$$\bar{L}^*(d) = \mathrm{E}^*[\bar{L}(d)],$$

and the optimal decision M^* minimises this expected loss. With analogous definitions of $\bar{L}^*_J(d, x_J)$ and $M^*_J(x_J)$ when we learn the value of x_J, we can define the expected value of information

$$V^*_J = \bar{L}^*(M^*) - \mathrm{E}[\bar{L}^*_J(M^*_J(X_J), X_J)].$$

However, this analysis is for decision making in the presence of code uncertainty. It implicity supposes that we learn about x_J without learning any more about the simulator, i.e. it is a value of information for reducing input uncertainty without reducing code uncertainty. If we are computing value of information measures for the purpose of prioritising research then it would be right to ignore any additional code uncertainty, because the computational cost of making enough simulator runs to render code uncertainty negligible will typically be much less than any research cost to reduce input uncertainty.

In this context, we are concerned with code uncertainty about a decision. First define the expected value of eliminating code uncertainty as

$$V^* = \bar{L}^*(M^*) - \mathrm{E}^*[\bar{L}(M)].$$

Then the expected value of information from learning the value of X_J becomes simply $\mathrm{E}^*[V_J]$. If we can also calculate $\mathrm{Var}^*[V_J]$ then this will quantify code uncertainty about that value of information.

SA in toolkit threads

The core, variant and generic threads in the toolkit are designed to take the reader through the process of building an emulator and then through to using it for various standard tasks, one of which is SA. So in principle procedures for carrying out SA should feature in each of these threads. In practice, not all of them yet have detailed SA procedures, although in due course it is the intention for all to do so.

- In the core thread for GP emulators, *ThreadCoreGP*, there are detailed procedures for variance based SA in *ProcVarSAGP*.
- In the core thread for BL emulators, *ThreadCoreBL* refers also to *ProcVarSAGP*. In Bayes linear theory it is more usual to express knowledge about uncertain quantities by means, variances and covariances, rather than complete distributions. So although the use of a full probability distribution $\omega(x)$ for the inputs, which is needed for probabilistic SA, is legitimate within BL theory (and allows *ProcVarSAGP* to be used) it does not fit so comfortably in the BL thread.
- In the variant thread dealing with multiple outputs, *ThreadVariantMultipleOutputs*, there are as yet no detailed SA procedures.
- In the variant thread dealing with dynamic emulators, *ThreadVariantDynamic*, there are as yet no detailed SA procedures.
- In the variant thread dealing with two-level emulation, *ThreadVariantTwoLevelEmulation*, the emphasis is on using many runs of a cheap simulator to construct prior information. The result is an emulator built according to the core GP or BL methods, so that the relevant details are given again in *ProcVarSAGP*.
- In the variant thread dealing with using observed derivatives, *ThreadVariantWithDerivatives*, there are as yet no detailed procedures for SA.
- In the generic thread dealing with emulating derivatives, *ThreadGenericEmulateDerivatives*, it is remarked that derivatives are used for local SA (see *DiscWhyProbabilisticSA*).
- In the generic thread dealing with combining multiple emulators, *ThreadGenericMultipleEmulators*, there are as yet no detailed procedures for SA.

8.109.3 Additional comments

Although none of the threads has any procedures for decision based SA, it is hoped to address this topic in some threads in future.

8.110 Discussion: Nugget effects in uncertainty and sensitivity analyses

8.110.1 Description and Background

The possibility of including a *nugget* term in the correlation function for a *Gaussian process emulator* is discussed in the alternatives page on emulator prior correlation function (*AltCorrelationFunction*). In the procedure pages for uncertainty analysis (*ProcUAGP*) and variance based sensitivity analysis (*ProcVarSAGP*) using a GP emulator, concerned respectively with *uncertainty analysis* and *variance based sensitivity analysis* for the *core problem*, closed form expresssions are given for various integrals for some cases where a nugget term may be included. This page provides a technical discussion of the reason for some apparent anomalies in those formulae.

8.110.2 Discussion

The closed form expressions are derived for the generalised Gaussian correlation function with nugget (see *AltCorrelationFunction*)

$$c(x,x') = \nu I_{x=x'} + (1-\nu)\exp\{-(x-x')^T C(x-x')\}$$

where $I_{x=x'}$ equals 1 if $x = x'$ but is otherwise zero, and ν represents the nugget term. They also assume a normal distribution for the uncertain inputs.

In both *ProcUAGP* and *ProcVarSAGP* a number of integrals are computed depending on $c(x,x')$, and we might expect the resulting integrals in each case to have two parts, one multiplied by ν and the other multiplied by $(1-\nu)$. In most cases, however, only the second term is found.

The reason for this is the fact that the nugget only arises when the two arguments of the correlation function are equal. The integrals all integrate with respect to the distribution of the uncertain inputs x, and since this is a normal distribution it gives zero probability to x equalling any particular value. We therefore find that in almost all of these integrals the nugget only appears in the integrand for a set of probability zero, and so it does not appear in the result of evaluating that integral. The sole exception is the integral denoted by U_p, where the nugget does appear leading to a result $U_p = \nu + (1-\nu) = 1$.

The heuristic explanation for this is that a nugget term is a white noise addition to the smooth simulator output induced by the generalised Gaussian correlation term. If we average white noise over even a short interval the result is necessarily zero because in effect we are averaging an infinite number of independent terms.

The practical implication is that the nugget reduces the amount that we learn from the training data, and also reduces the amount that we could learn in sensitivity analysis calculations by learning the values of a subset of inputs.

8.111 Discussion: Uncertainty analysis

In uncertainty analysis, we wish to quantify uncertainty about simulator outputs due to uncertainty about simulator inputs. We define X to be the uncertain true inputs, and $f(X)$ to be the corresponding simulator output(s). In the emulator framework, $f(.)$ is also treated as an uncertain function, and it is important to consider both uncertainty in X and uncertainty in $f(.)$ when investigating uncertainty about $f(X)$. In particular, it is important to distinguish between the unconditional distribution of $f(X)$, and the distribution of $f(X)$ conditional on $f(.)$. For example:

1. $\mathrm{E}[f(X)]$ is the expected value of $f(X)$, where the expectation is taken with respect to both $f(.)$ and X. The value of this expectation can, in principle, be obtained for any emulator and input distribution.
2. $\mathrm{E}[f(X)|f(.)]$ is the expected value of $f(X)$, where the expectation is taken with respect to X only as $f(.)$ is given. If $f(.)$ is a computationally cheap function, we could, for example, obtain the value of this expectation

> using Monte Carlo, up to an arbitrary level of precision. However, when $f(.)$ is computationally expensive such that we require an emulator for $f(.)$, **this expectation is an uncertain quantity**. We are uncertain about the value of $\mathrm{E}[f(X)|f(.)]$, because we are are uncertain about $f(.)$.

There is no sense in which $\mathrm{E}[f(X)]$ can be 'wrong': it is simply a probability statement resulting from a choice of emulator (good or bad) and input distribution. But an estimate of $\mathrm{E}[f(X)|f(.)]$ obtained using an emulator can be poor if we have a poor emulator (in the *validation* sense) for $f(.)$. Alternatively, we may be very uncertain about $\mathrm{E}[f(X)|f(.)]$ if we don't have sufficient training data for the emulator of $f(.)$. Hence in practice, the distinction is important for considering *whether we have enough simulator runs for our analysis of interest*.

8.112 Discussion: Variance-based Sensitivity Analysis

8.112.1 Description and background

The basic ideas of *sensitivity analysis* (SA) are presented in the topic thread on sensitivity analysis (*ThreadTopicSensitivityAnalysis*). We concentrate in the *MUCM* toolkit on probabilistic SA, but the reasons for this choice and alternative approaches are considered in the discussion page *DiscWhyProbabilisticSA*. In probabilistic SA, we assign a function $\omega(x)$ that is formally treated as a probability density function for the inputs. The interpretation and specification of $\omega(x)$ is considered in the context of specific uses of SA; see the discussion pages on SA measures for simplification (*DiscSensitivityAndSimplification*) and SA measures for output uncertainty (*DiscSensitivityAndOutputUncertainty*).

Variance-based sensitivity analysis is a form of probabilistic sensitivity analysis in which the impact of an input or a group of inputs on a *simulator*'s output is measured through variance and reduction in variance. We develop here the principal measures of influence and sensitivity for individual inputs and groups of inputs. Our development is initially in terms of a single simulator output, before moving on to consider multiple outputs.

Notation

The following notation and terminology is introduced in *ThreadTopicSensitivityAnalysis*.

In accordance with the standard *toolkit notation*, we denote the simulator by f and its inputs by x. The focus of SA is the relationship between x and the simulator output(s) $f(x)$. Since SA also typically tries to isolate the influences of individual inputs, or groups of inputs, on the output(s), we let x_j be the j-th element of x and will refer to this as the j-th input, for $j = 1, 2, \ldots, p$, where as usual p is the number of inputs. If J is a subset of the indices $\{1, 2, \ldots, p\}$, then x_J will denote the corresponding subset of inputs. For instance, if $J = \{2, 6\}$ then $x_J = x_{\{2,6\}}$ comprises inputs 2 and 6, while x_j is the special case of x_J when $J = \{j\}$. Finally, x_{-j} will denote the whole of the inputs x *except* x_j, and similarly x_{-J} will be the set of all inputs except those in x_J.

Formally, $\omega(x)$ is a joint probability density function for all the inputs. The marginal density function for input x_j is denoted by $\omega_j(x_j)$, while for the group of inputs x_J the density function is $\omega_J(x_J)$. The conditional distribution for x_{-J} given x_J is $\omega_{-J|J}(x_{-J}\,|\,x_J)$. Note, however, that it is common for the distribution ω to be such that the various inputs are statistically independent. In this case the conditional distribution $\omega_{-J|J}$ does not depend on x_J and is identical to the marginal distribution ω_{-J}.

Note that by assigning probability distributions to the inputs we formally treat those inputs as random variables. Notationally, it is conventional in statistics to denote random variables by capital letters, and this distinction is useful also in probabilistic SA. Thus, the symbol X denotes the set of inputs when regarded as random (i.e. uncertain), while x continues to denote a particular set of input values. Similarly, X_J represents those random inputs with subscripts in the set J, while x_J denotes an actual value for those inputs.

8.112.2 Discussion

We present here the basic variance based SA measures. Some more technical details may be found in the discussion page on the theory of variance based SA (*DiscVarianceBasedSATheory*).

The mean effect $M_J(x_J)$

If we wish to see how varying the input x_j influences the simulator output $f(x)$, we could choose values of all the other inputs x_{-j} and run the simulator with these values fixed and with a variety of different values of x_j. However, the response of $f(x)$ to varying x_j will generally depend on the values we choose for x_{-j}. We therefore define the *mean effect* of x_j to be the average value of $f(x)$, averaged over the possible values of x_{-j}. The appropriate averaging is to take the expectation with respect to the conditional distribution of X_{-j} given x_j. We denote the mean effect by

$$M_j(x_j) = \mathrm{E}[f(X)\,|\,x_j] = \int f(x)\,\omega_{-j|j}(x_{-j}\,|\,x_j)\,dx_{-j}.$$

This is a function of x_j, and represents an average response of the simulator output to varying x_j. Simply plotting this function gives a visual impression of how x_j influences the output.

More generally, we can define the mean effect of a group of inputs:

$$M_J(x_J) = \mathrm{E}[f(X)\,|\,x_J] = \int f(x)\,\omega_{-J|J}(x_{-J}\,|\,x_J)\,dx_{-J}.$$

The mean effect of x_J is a function of x_J, and so is more complex than the mean effect of a single input. However, it can reveal more structure than the main effects of the individual inputs in the group.

Interactions and main effects $I_J(x_J)$

Let the average effect of changing x_1 from x_1^0 to x_1^1 be denoted by $a_1 = M_1(x_1^1) - M_1(x_1^0)$, and similarly let the average effect of changing x_2 from x_2^0 to x_2^1 be denoted by $a_2 = M_2(x_2^1) - M_2(x_2^0)$. Now consider changing both of these inputs, from (x_1^0, x_2^0) to (x_1^1, x_2^1). For a simple, smooth simulator, we might think that the average effect of such a change might be $a_1 + a_2$. However, this will generally not be the case. The actual average change will be $M_{\{1,2\}}(x_1^1, x_2^1) - M_{\{1,2\}}(x_1^0, x_2^0)$, and where this is different from $a_1 + a_2$ there is said to be an interaction between x_1 and x_2.

Formally, the interaction effect between inputs x_j and $x_{j'}$ is defined to be

$$I_{\{j,j'\}}(x_{\{j,j'\}}) = M_{\{j,j'\}}(x_{\{j,j'\}}) - I_j(x_j) - I_{j'}(x_{j'}) - M\,, \qquad (1)$$

where

$$M = \mathrm{E}[f(X)] = \int f(x)\,\omega(x)dx$$

is the overall expected value of the simulator output, and

$$I_j(x_j) = M_j(x_j) - M\,. \qquad (2)$$

These definitions merit additional explanation! First, M is the uncertainty mean, which is one of the quantities typically computed in *uncertainty analysis*. For our purposes it is a reference point. If we do not specify the values of any of the inputs, then M is the natural estimate for $f(X)$.

If, however, we specify the value of x_j, then the natural estimate for $f(X)$ becomes $M_j(x_j)$. We can think of this as the reference point M plus a deviation $I_j(x_j)$ from that reference point. We call $I_j(x_j)$ the *main effect* of x_j.

(It is easy to confuse the mean effect $M_j(x_j)$ with the main effect $I_1(x_1)$ - the two terms are obviously very similar - and indeed some writers call $M_j(x_j)$ the main effect of x_j. In informal discussion, such imprecision in terminology is unimportant, but the distinction is useful in formal analysis and we will endeavour to use the terms "mean effect" and "main effect" precisely.)

Now if we specify both x_j and $x_{j'}$, equation (1) expresses the natural estimate $M_{\{j,j'\}}(x_{\{j,j'\}})$ as the sum of (a) the reference point M, (b) the two main effects $I_j(x_j)$ and $I_{j'}(x_{j'})$, and (c) the interaction effect $I_{\{j,j'\}}(x_{\{j,j'\}})$.

We can extend this approach to define interactions of three or more inputs. The formulae become increasingly complex and the reader may choose to skip the remainder of this subsection because such higher-order interactions are usually quite small and are anyway hard to visualise and interpret.

For the benefit of readers who wish to see the detail, however, we define increasingly complex interactions recursively via the general formula

$$I_J(x_J) = M_J(x_J) - \sum_{J' \subset J} I_{J'}(x_{J'})\,, \qquad (3)$$

where the sum is over all interactions for which J' is a proper subset of J. Notice that when J' contains only a single input index $I_{J'}(x_{J'})$ is a main effect, and by convention we include in the sum the case where J' is the empty set whose "interaction" term is just M.

By setting $J = \{j\}$, equation (3) gives the main effect definition (2), and with $J = \{j.j'\}$ it gives the two-input interaction definition (1). A three-input interaction is obtained by taking the corresponding three-input mean effect and subtracting from it (a) the reference point M, (b) the three single-input main effects and (c) the three two-input interactions. And so on.

The sensitivity variance V_J

Mean interactions are quite detailed descriptions of the effects of individual inputs and groups of inputs, because they are functions of those inputs. For many purposes, it is helpful to have a single figure summary of how sensitive the output is to a given input. Does that input have a "large" effect on the output? (The methods developed in the discussion page on decision based SA (*DiscDecisionBasedSA*) are different from those derived here because there we ask the question whether an input has an "important" effect, in terms of influencing a decision.)

We measure the magnitude of an input's influence on the output by the expected square of its main effect, or equivalently by the variance of its mean effect. This definition naturally applies also to a group, so we define generally

$$V_J = \mathrm{Var}[M_J(X_J)] = \int \{M_J(x_J) - M\}^2\, \omega_J(x_J)\, dx_J.$$

This is called the sensitivity variance of x_J. Although we have derived this measure by thinking about how large an effect, on average, varying the inputs x_J has on the output, there is another very useful interpretation.

Consider the overall variance

$$V = \mathrm{Var}[f(X)] = \int \{f(x) - M\}^2\, \omega(x)\, dx.$$

This is another measure that is commonly computed as part of an uncertainty analysis. It expresses the overall uncertainty about $f(X)$ when X is uncertain (and has the probability distribution defined by $\omega(x)$). If we were to learn the correct values of the inputs comprising x_J, then we would expect this to reduce uncertainty about $f(X)$. The variance conditional on learning x_J would be

$$w(x_J) = \mathrm{Var}[f(X)\,|\,x_J] = \int \{f(x) - M_J(x_J)\}^2\, \omega_{-J|J}(x_{-J}\,|\,x_J)\, dx_{-J}.$$

Notice that this would depend on what value we discovered x_J had, which of course we do not know. A suitable measure of what the uncertainty would be after learning x_J is the expected value of this conditional variance, i.e.

$$W_J = \mathrm{E}[w(X_J)] = \int w(x_J)\, \omega_J(x_J)\, dx_J.$$

It is shown in *DiscVarianceBasedSATheory* that V_J is the amount by which uncertainty is reduced, i.e.

$$V_J = V - W_J.$$

Therefore we have two useful interpretations of the sensitivity variance, representing different ways of thinking about the sensitivity of $f(x)$ to inputs x_J:

1. V_J measures the average magnitude of the mean effect $M_J(x_J)$, and so describes the scale of the influence that x_J has on the output, on average.
2. V_J is the amount by which uncertainty about the output would be reduced, on average, if we were to learn the correct values for the inputs in x_J, and so describes how much of the overall uncertainty is due to uncertainty about x_J.

The second of these is particularly appropriate when we are using SA to analyse uncertainty about the simulator output that is induced by uncertainty in the inputs. The first is often more relevant when we are using SA to understand the simulator (when $\omega(x)$ may be simply a weight function rather than genuinely a probability density function).

The sensitivity index S_J and total sensitivity index T_J

The sensitivity variance V_J is in units that are the square of the units of the simulator output, and it is common to measure sensitivity instead by a dimensionless index. The *sensitivity index* of x_J is its sensitivity variance V_J expressed as a proportion of the overall variance V:

$$S_J = V_J/V.$$

Thus an index of 0.5 indicates that uncertainty about x_J accounts for half of the the overall uncertainty in the output due to uncertainty in x. (The index is often multiplied by 100 so as to be expressed as a percentage; for instance an index of 0.5 would be referred to as 50%.)

However, there is another way to think about how much of the overall uncertainty is attributable to x_J. Instead of considering how much uncertainty is reduced if we were to learn x_J, we could consider how much uncertainty *remains* after we learn the values of all the other inputs, which is W_{-J}. This is not the same as V_J, and in most situations is greater. So another index for S_J is its *total sensitivity index*

$$T_J = W_{-J}/V = 1 - S_{-J}.$$

The variance partition theorem

The relationship between these indices (and the reason why T_J is generally larger than S_J) can be seen in a general theorem that is proved in *DiscVarianceBasedSATheory*. First notice that for an individual x_j the sensitivity variance V_j is not just the variance of the mean effect $M_j(X_j)$ but also of its main effect $I_j(X_j)$ (since the main effect is just the mean effect minus a constant). However, it is not true that for a group x_J of more than one input V_J equals the variance of $I_J(X_J)$. If we define

$$V_J^I = \mathrm{Var}[I_J(X_J)]$$

then we have $V_j^I = V_j$ but otherwise we refer to V_J^I as the interaction variance of x_J. Just as an interaction effect between two inputs x_j and $x_{j'}$ concerns aspects of the joint effect of those two inputs that are not explained by their main effects alone, their interaction variance concerns uncertainty that is attributable to the joint effect that is not explained by their separate sensitivity variances. Interaction variances are a useful summary of the extent of interactions between inputs.

Specifically, the following result holds (under a condition to be presented shortly)

$$V_{\{j,j'\}} = V_j + V_{j'} + V^I_{\{j,j'\}}.$$

So the amount of uncertainty attributable to (and removed by learning) both x_j and $x_{j'}$ is the sum of the amounts attributable to each separately (their separate sensitivity variances) plus their interaction variance.

The condition for this to hold is that x_j and $x_{j'}$ must be statistically independent. This independence is a property that can be verified from the probability density function $\omega(x)$.

Generalising, suppose that all the inputs are mutually independent. This means that their joint density function $\omega(x)$ reduces to the product $\prod_{j=1}^p \omega_j(x_j)$ of their marginal density functions. Independence often holds in practice, partly because it is much easier to specify $\omega(x)$ by thinking about the uncertainty in each input separately.

If the :math:‘x_j‘s are mutually independent, then

$$V_J = \sum_{J' \subseteq J} V^I_{J'} \,. \qquad (4)$$

Thus the sensitivity variance of a group x_J is the sum of the individual sensitivity variances of the inputs in the group plus all the interaction variances between members of the group. (Notice that this time the sum is over all J' that are subsets of J including J itself.)

In particular, the total variance can be partitioned into the sum of all the sensitivity and interaction variances:

$$V = \sum_J V^I_J.$$

This is the partition theorem that is proved in *DiscVarianceBasedSATheory*. We now consider what it means for the relationship between S_J and T_J.

First consider S_J. From the above equation (4), this is the sum of the individual sensitivity indices S_j for inputs x_j in X_J, plus all the interaction variances between the inputs in x_J, also expressed as proportions of V.

On the other hand T_J can be seen to be the sum of the individual sensitivity indices S_j plus all the interaction variances (divided by V) that are *not* between inputs in x_{-J}. The difference is subtle, perhaps, but the interactions whose variances are included in T_j are all the ones included in S_J plus interactions that are between (one or more) inputs in x_J and (one or more) inputs outside x_J. This is why T_J is generally larger than S_J.

The difference between T_J and S_J is in practice an indicator of the extent to which inputs in x_J interact with inputs in x_{-J}. In particular, the difference between T_j and S_j indicates the extent to which x_j is involved in interactions with other inputs.

Regression variances and coefficients

Another useful group of measures is associated with fitting simple regression models to the simulator. Consider approximating $f(x)$ using a regression model of the form

$$\hat{f}_g(x) = \alpha + g(x)^{\mathrm{T}}\gamma$$

where $g(x)$ is a chosen vector of fitting functions and α and γ are parameters to be chosen for best fit (in the sense of minimising expected squared error with respect to the distribution :math:omega(x)). The general case is dealt with in *DiscVarianceBasedSATheory<DiscVarianceBasedSATheory>*; here we outline just the simplest, but in many ways the most useful, case.

Consider the case where $g(x) = x_j$. Then the best fitting value of γ defines an average gradient of the effect of x_j, and is given by

$$\gamma_j = \mathrm{Cov}[X_j, f(X)]/\mathrm{Var}[X_j].$$

Using this fitted regression reduces uncertainty about $f(X)$ by an amount

$$V^L_j = \mathrm{Cov}[X_j, f(X)]^2/\mathrm{Var}[X_j].$$

This can be compared with the sensitivity variance of x_j, which is the reduction in uncertainty that is achieved by learning the value of x_j. The sensitivity variance V_j will always be greater than or equal to V_j^L, and the difference between the two is a measure of the degree of nonlinearity in the effect of x_j.

Notice that the variance and covariance needed in these formulae are evaluated using the distribution $\omega(x)$. So $\mathrm{Var}[X_j]$ is just the variance of the marginal distribution of x_j, and if $\bar{x}_j$ is the mean of that marginal distribution then

$$\mathrm{Cov}[X_j, f(X)] = \int x_j\, M_j(x_j)\, \omega_j(x_j)\, dx_j - \bar{x}_j\, M.$$

Multiple outputs

If $f(x)$ is a vector of r outputs, then all of the above measures generalise quite simply. The mean and main effects are now $r \times 1$ vector functions of their arguments, and all of the variances become $r \times r$ matrices. For example, the the overall variance becomes the matrix

$$V = \mathrm{Var}[f(X)] = \int \{f(x) - M\}\{f(x) - M\}^{\mathrm{T}}\, \omega(x)\, dx.$$

Note, however, that we do not consider matrix versions of S_J and T_J, because it is not really meaningful to divide a sensitivity variance matrix V_J by the overall variance matrix V.

In practice, there is little extra understanding to be obtained by attempting an SA of multiple outputs in this way beyond what can be gained by SA of each output separately. Often, if the primary interest is not in a single output then it can be defined in terms of a single function of the outputs, and then SA carried out on that single function is indicated. Some more discussion of SA on a function of $f(x)$, with an example of SA applied to whether $f(x)$ exceeds some threshold, may be found in *DiscSensitivityAndOutputUncertainty*.

8.113 Discussion: Theory of variance-based sensitivity analysis

8.113.1 Description and background

The methods of variance-based *sensitivity analysis* are set out in the discussion page *DiscVarianceBasedSA*. Some technical results and details are considered here.

Notation and terminology may all be found in *DiscVarianceBasedSA*, although we often repeat definitions here for clarity.

8.113.2 Discussion

Equivalent interpretations of V_J

In *DiscVarianceBasedSA*, the sensitivity variance V_J for inputs x_J is defined as $\mathrm{Var}[M_J(X_J)]$, where $M_J(x_J)$ is the mean effect of x_J, defined as $\mathrm{E}[f(X)\,|\,x_J]$. Now the following is a general result in Statistics: let Y and Z be two random variables, then

$$\mathrm{Var}[Y] = \mathrm{E}[\mathrm{Var}[Y\,|\,Z]] + \mathrm{Var}[\mathrm{E}[Y\,|\,Z]].$$

In words, the variance of Y can be expressed as a sum of two parts, the first is the expectation of the conditional variance of Y given Z, and the second is the variance of the conditional expectation of Y given Z.

We use this formula by equating Y to $f(X)$ and Z to X_J. From the above definitions, the second term on the right hand side of the formula is seen to be V_J. The other terms easily reduce to other measures that are defined in

DiscVarianceBasedSA. The left hand side is $\mathrm{Var}[f(X)]$, which is the overall output variance V. The first term on the right hand side is the expectation of $\mathrm{Var}[f(X)\,|\,X_J]$, which is $w(X_J)$. We therefore have the result

$$V = W_J + V_J.$$

This provides the second interpretation of V_J in *DiscVarianceBasedSA*. It is the amount by which uncertainty is expected to be reduced if we were to learn x_J, i.e. $V - W_J$.

The variance partition theorem for independent inputs

A very useful result in variance-based SA theory is equation (4) in *DiscVarianceBasedSA*, and concerns interaction variances when the inputs x_j (for $j = 1, 2, \ldots, p$) in x are all mutually independent. In *DiscVarianceBasedSA*, equation (3) can be rewritten as

$$M_J(x_J) = \sum_{J' \subseteq J} I_{J'}(x_{J'})\,. \qquad (A)$$

In this formula, the sum is over all subsets J' of J, including the empty set and J itself. The term $I_{J'}(x_{J'})$ is in general called the interaction effect of $x_{J'}$, except that when J' contains just a single element $J' = \{j\}$ then it is referred to as the main effect of x_j, and when J' is the empty set we define it to be the uncertainty mean $M = \mathrm{E}[f(X)]$.

Example: Suppose that $J = \{1, 2, 3\}$, then there are eight possible subsets J' and equation (A) becomes

$$\begin{aligned} M_{\{1,2,3\}}(x_{\{1,2,3\}}) = \quad & I_{\{1,2,3\}}(x_{\{1,2,3\}}) \\ & + I_{\{1,2\}}(x_{\{1,2\}}) + I_{\{1,3\}}(x_{\{1,3\}}) + I_{\{2,3\}}(x_{\{2,3\}}) \\ & + I_{\{1\}}(x_{\{1\}}) + I_{\{2\}}(x_{\{2\}}) + I_{\{3\}}(x_{\{3\}}) + M. \end{aligned}$$

The first term in this example is the three-input interaction; on the next line are the three two-input interactions; on the third line are the three main effects and finally the overall mean.

Now consider V_J, which is the variance of $M_J(X_J)$. Taking the variance of the right hand side of (A), a standard result in Statistics is that the variance of a sum of random variables is the sum of the variances of those random variables plus the sum of the covariances between all pairs. Now it can be shown that when the inputs are independent all those covariances are zero. Since the variances of the interaction terms in (A) are the interaction variances V^I_J this yields equation (4) in *DiscVarianceBasedSA*.

In order to prove that result, it remains to show that all covariances are zero when inputs are independent. First note that because inputs are independent the covariance between $I_J(X_J)$ and $I_{J'}(X_{J'})$ will be zero if J and J' have no elements in common. To illustrate the proof when there are elements in common, consider the correlation between the main effect

$$I_{\{j\}}(X_{\{j\}}) = M_{\{j\}}(X_{\{j\}}) - M$$

and the two-input interaction

$$I_{\{j,j'\}}(X_{\{j,j'\}}) = M_{\{j,j'\}}(X_{\{j,j'\}}) - M_{\{j\}}(X_{\{j\}}) - M_{\{j'\}}(X_{\{j'\}}) + M.$$

First notice that $\mathrm{E}[I_{\{j\}}(X_{\{j\}})] = M - M = 0$ and $\mathrm{E}[I_{\{j,j'\}}(X_{\{j,j'\}})] = M - M - M + M = 0$.

In fact, the expectation of every interaction term $I_J(X_J)$ is zero when J is not the empty set. However, this expectation is taken with respect to all of X_J; it is also true that if we take the expectation with respect to just one X_j, for any j in J the result is also zero *provided* that the inputs are independent. For instance, in the case of $I_{\{j,j'\}}(X_{\{j,j'\}})$ consider the expectation with respect to X_j only. If X_j and $X_{j'}$ are independent, we obtain $M_{\{j'\}}(X_{\{j'\}}) - M - M_{\{j'\}}(X_{\{j'\}}) + M = 0$.

Now we can use these facts to find the covariance. First, because the expectations are zero the covariance equals the expectation of the product $I_{\{j\}}(X_{\{j\}}) \times I_{\{j,j'\}}(X_{\{j,j'\}})$. This is the expectation with respect to both X_j and $X_{j'}$.

Because of independence, we can take the expectation in two stages, first with respect to $X_{j'}$ then with respect to X_j. However, the first step produces zero because the expectation of $I_{\{j,j'\}}(X_{\{j,j'\}})$ with respect to either of the inputs is zero. Hence the covariance is zero.

The same argument works for any covariance. The covariance will be the expectation of the product, and that will be zero when we first take expectation with respect to any input that is in one but not both of X_J and $X_{J'}$.

Non-independent inputs

When the inputs are not independent, the above theorem does not hold. As a result, some convenient simplifications also fail. For simplicity, suppose we have just two inputs, X_1 and X_2. With independence, we have

$$V = V_{\{1,2\}} = V_{\{1\}} + V_{\{2\}} + I_{\{1,2\}}$$

If independence does not hold, then $V_{\{1\}}$, $V_{\{2\}}$ and $I_{\{1,2\}}$ are still defined as before and are positive quantities, but their sum will not generally be V. Furthermore, in the case of independence, the total sensitivity index $T_{\{1\}}$ of X_1 will be larger than its sensitivity index $S_{\{1\}}$, but this is also not necessarily true when inputs are not independent.

Consider the case where the simulator has the form $f(x_1, x_2) = x_1 + 2x_2$, and suppose that X_1 and X_2 both have zero means and unit variances but have correlation ρ. Now we find after some simple algebra that $V = 2(1+\rho)$, $V_{\{1\}} = V_{\{2\}} = (1+\rho)^2$ and $I_{\{1,2\}} = 2\rho^2(1+\rho)$. The last of these is perhaps surprising. When the simulator is strictly a sum of two terms, one a function of x_1 and the other a function of x_2, we would think of this as a case where there is no interaction, and indeed if the inputs were independent then $I_{\{1,2\}}$ would be zero, but this is not true when they are correlated.

For this example we have sensitivity indices

$$S_{\{1\}} = S_{\{2\}} = (1+\rho)/2\,, \qquad T_{\{1\}} = T_{\{2\}} = (1-\rho)/2,$$

and clearly $T_{\{1\}} < S_{\{1\}}$ in this instance if the correlation is positive. 'Total sensitivity' can be a misleading term.

When the inputs are not independent it is not generally helpful to look at interactions, and the sensitivity and total sensitivity indices must be interpreted carefully.

Sequential decomposition

The following way of decomposing the total uncertainty variance V applies whether the inputs are independent or not, and is sometimes convenient computationally.

For exposition, suppose that we have just four uncertain inputs, $X = (X_1, X_2, X_3, X_4)$. Also, to clarify the formulae, we add subscripts to the expectation and variance operators to denote which distribution they are applied to.

$$\begin{aligned} V = \mathrm{Var}_X[y(X)] = &\ \mathrm{Var}_{X_1}[\mathrm{E}_{X_2,X_3,X_4|X_1}[y(X)]] + \mathrm{E}_{X_1}[\mathrm{Var}_{X_2,X_3,X_4|X_1}[y(X)]] \\ = &\ \mathrm{Var}_{X_1}[\mathrm{E}_{X_2,X_3,X_4|X_1}[y(X)]] + \mathrm{E}_{X_1}[\mathrm{Var}_{X_2|X_1}[\mathrm{E}_{X_3,X_4|X_1,X_2}[y(X)]]] \\ &\quad + \mathrm{E}_{X_1,X_2}[\mathrm{Var}_{X_3,X_4|X_1,X_2}[y(X)]] \\ = &\ \mathrm{Var}_{X_1}[\mathrm{E}_{X_2,X_3,X_4|X_1}[y(X)]] + \mathrm{E}_{X_1}[\mathrm{Var}_{X_2|X_1}[\mathrm{E}_{X_3,X_4|X_1,X_2}[y(X)]]] \\ &\quad + \mathrm{E}_{X_1,X_2}[\mathrm{Var}_{X_3|X_1,X_2}[\mathrm{E}_{X_4|X_1,X_2,X_3}[y(X)]]] + \mathrm{E}_{X_1,X_2,X_3}[\mathrm{Var}_{X_4|X_1,X_2,X_3}[y(X)]] \end{aligned}$$

The formula can obviously be continued for more inputs, and we can replace individual inputs with groups of inputs.

If inputs are independent, then the conditioning in the subscripts can be removed. In this case, note that the first term in the decomposition is $V_{\{1\}}$, the second term, however, is $V_{\{2\}} + I_{\{1,2\}}$. Each successive term includes interactions with the inputs coming earlier in the sequence. Finally, the last term is $T_{\{4\}}$.

If inputs are not independent, then this is a useful decomposition that does not rely on separately identifying interactions (which we have seen are not helpful in this case). However, it is also clear that the decomposition depends on the order of the inputs, and we can obviously define sequential decompositions in other sequences. This is analgous to the analysis of variance in conventional statistics theory when factors are not orthogonal.

Regression variances

In the page discussing various forms of SA (*DiscWhyProbabilisticSA*), one of those forms is regression SA. In this approach, a linear regression function is fitted to the simulator output $f(x)$ and sensitivity measures are calculated from the resulting fitted regression coefficients.

Consider such a regression fit in which the regressor variables are denoted by a vector $g(x)$ of functions of the inputs. The fitted approximation to the simulator is $\hat{f}(x) = \hat{\gamma}^{\rm T} g(x)$, where $\hat{\gamma}$ is the corresponding vector of fitted coefficients. This plays the same role as the mean effect $M(x)$, and from the perspective of variance-based SA, we can define a corresponding sensitivity variance

$$V_g = {\rm Var}(\hat{\gamma}^{\rm T} g(X)).$$

This will always be less than the overall variance V but will get closer to it the more accurately the fitted regression is able to approximate the true mean effect. Similarly, if $g(x)$ is a function only of X_J, then $V_g < V_J$ and the difference between the two is smaller the more closely the regression function can approximate $M_J(x)$.

This provides one nice use for regression variances. Let $g(x)^{\rm T} = (1, x_j)$, so that the regression fit is a simple straight line in x_j. We call the resulting regression variance the linear sensitivity variance of x_j, and the difference between this and $V_{\{j\}}$ is an indicator of how nonlinear the mean effect of x_j is.

8.113.3 Additional comments

Formal definitions of regression variances and theory for computing them from emulators is given in

Oakley, J. E. and O'Hagan, A. (2004). Probabilistic sensitivity analysis of complex models: a Bayesian approach. *Journal of the Royal Statistical Society B* 66, 751-769.

8.114 Discussion: Why Model Discrepancy?

8.114.1 Description and Background

Often, the purpose of a model, $f(x)$, of a physical process, is to make statements about the corresponding real world process y (see the variant thread on linking models to reality using model discrepancy *ThreadVariantModelDiscrepancy*). In order for this to be possible, the difference between the model and the real system must be *assessed* and incorporated into the analysis. This difference is known as the *model discrepancy*, d, and this page discusses the reasons why such a quantity is essential. Here we use the term model synonymously with the term *simulator*.

8.114.2 Discussion

A model is an imperfect representation of the way system properties affect system behaviour. Whenever models are used, this imperfection should be accounted for within our analysis, by careful assessment of Model Discrepancy.

Model discrepancy is a feature of computer model analysis which is relatively unfamiliar to most scientists. However carefully we have constructed our model, there will always be a difference between the system and the simulator. A climate model will never be precisely the same as climate, and nor would we expect it to be. Inevitably, there will be simplifications in the physics, based on features that are too complicated for us to include, features that we do not know that we should include, mismatches between the scales on which the model and the system operate, simplifications and approximations in solving the equations determining the system, and uncertainty in the forcing functions, boundary conditions and initial conditions.

Incorporating the difference between the model and the real system within our analysis will allow us

- to make meaningful predictions about future behaviour of the system, rather than the model,
- to perform a proper *calibration* of the model inputs to historical observation.

Obviously we will be uncertain about this difference, d, and it is therefore natural to express such judgements probabilistically, to be incorporated within a Bayesian analysis, employing either the fully probabilistic or the Bayes Linear approach.

Neglecting the model discrepancy would make all further statements in the analysis conditional on the model being a perfect representation of reality, and would result in overconfident predictions of system behaviour. In fact, ignoring d leads to a smaller variance of (and hence overconfidence in) the observed data z, which leads to over-fitting in calibration (when we learn about x). This exacerbates the over-confidence effect in subsequent predictions. Similarly, ignoring the correlation structure within d leads to further prediction inaccuracies. Fundamental to the scientific process is the concern that without d, one can wrongly rule out a potentially useful model. Unfortunately, all these mistakes are commonly made.

The simplest and most common strategy for incorporating model discrepancy is known as the *Best Input* Approach, which is discussed in detail in page *DiscBestInput*, and gives a simple and intuitive definition of d. More careful methods have been developed that go beyond the limitations of the Best Input approach, one such approach involves the idea of Reification (see the discussion pages on reification (*DiscReification*) and its theory (*DiscReificationTheory*)).

Often, understanding the size and structure of the model discrepancy will be one of the most challenging aspects of the analysis. There are, however, a variety of methods available to assess d: for Expert, Informal and Formal methods see the discussion pages *DiscExpertAssessMD*, *DiscInformalAssessMD* and *DiscFormalAssessMD* respectively.

8.114.3 Additional Comments

The assessment of d can be a difficult and subtle task. We generally use the random quantity d to statistically model a difference that is, in reality, extremely complex. In this sense, there is no 'true' value of d itself, instead it should be viewed as a useful tool for representing an important feature (the difference between model and reality) in a simple and tractable manner. Before an assessment is made, we should be clear about which features of the model discrepancy we are interested in. For example, do we want to learn about the realised values of d itself, or do we wish to learn about the parameters of a distribution that d may be considered a realisation of. Such considerations are of particular importance in problems where the match between the model output and historical data is used to inform judgements about the likely values of model discrepancy for future system features.

8.115 Discussion: Why Probabilistic Sensitivity Analysis?

8.115.1 Description and background

The uses of *sensitivity analysis* (SA) are outlined in the topic thread on SA (*ThreadTopicSensitivityAnalysis*), where they are related to the methods of probabilistic SA that are employed in the *MUCM* toolkit. We discuss here alternative approaches to SA and why probabilistic SA is preferred.

First we review some notation and terminology that is introduced in *ThreadTopicSensitivityAnalysis*.

In accordance with the standard *toolkit notation*, we denote the *simulator* by f and its inputs by x. The focus of SA is the relationship between x and the simulator output(s) $f(x)$. Since SA also typically tries to isolate the influences of individual inputs, or groups of inputs, on the output(s), we let x_j be the j-th element of x and will refer to this as the j-th input, for $j = 1, 2, \ldots, p$, where as usual p is the number of inputs. If J is a subset of the indices $\{1, 2, \ldots, p\}$, then x_J will denote the corresponding subset of inputs. For instance, if $J = \{2, 6\}$ then $x_J = x_{\{2,6\}}$ comprises inputs 2 and 6, while x_j is the special case of x_J when $J = \{j\}$. Finally, x_{-j} will denote the whole of the inputs x *except* x_j, and similarly x_{-J} will be the set of all inputs except those in x_J.

8.115.2 Discussion

Local SA

SA began as a response to concern for the consequences of mis-specifying the values of inputs to a *simulator*. The simulator user could provide values $\hat{x}$ that would be regarded as best estimates for the inputs, and so $f(\hat{x})$ would in some sense be an estimate for the corresponding simulator output(s). However, if the correct inputs differed from $\hat{x}$ then the correct output would differ from $f(\hat{x})$. The output would be regarded as sensitive to a particular input x_j if the output changed substantially when x_j was perturbed slightly. This led to the idea of measuring sensitivity by differentiation of the function. The measure of sensitivity to x_j was the derivative $\partial f(x)/\partial x_j$, evaluated at $x = \hat{x}$.

SA based on derivatives has a number of deficiencies, however. The differential measures only the impact of an infinitesimal change in x_j, and for this reason this kind of SA is referred to as local SA. If the response of the output to x_j is far from linear, then perturbing x_j more than a tiny amount might have an effect that is not well represented by the derivative.

More seriously, the derivative is not invariant to the units of measurement. If, for instance, we choose to measure x_j in kilometres rather than metres, then $\partial f(x)/\partial x_j$ will change by a factor of 1000. The output may appear to be more sensitive to input x_j than to $x_{j'}$ because the derivative evaluated at $\hat{x}$ is larger, but this ordering could easily be reversed if we changed the scales of measurement.

One-way SA

Alternatives to local SA based on derivatives are known as global SA methods. A simple global method involves perturbing x_j from its nominal value $\hat{x}_j$, say to a point x'_j, with all other inputs held at their nominal values $\hat{x}_{-j}$. The resulting change in output is then regarded as a measure of sensitivity to x_j. This is known as one-way SA, because the inputs are varied only one at a time from their nominal values.

One-way SA addresses the problems noted for local SA. First, we do not consider only infinitesimal perturbations, and any nonlinearity in the response to x_j is accounted for in the evaluation of the output at the new point (where $x_j = x'_j$ but $x_{-j} = \hat{x}_{-j}$). Second, if we change the units of measurement for x_j then this will be reflected in both $\hat{x}_j$ and x'_j and the SA measure will be unaffected.

However, one-way SA has its own problems. The SA measures depend on how far we perturb the individual inputs, and the ordering of inputs produced by their SA measures can change if we change the x'_j values.

Also, one-way SA fails to quantify joint effects of perturbing more than one input together. For instance, we have measures for x_1 and x_2 but not for $x_{\{1,2\}}$.The effect of perturbing x_1 and x_2 together cannot be inferred from knowing the effects of perturbing them individually. Statisticians say that two inputs interact when the effect of perturbing both is not just the sum of the effects of perturbing them individually. One-way SA is not able to measure interactions.

Multi-way SA

In the wider context of experimental design, statisticians have for more than 50 years decried the idea of varying factors one at a time for precisely the reason that such an experiment cannot identify interactions between the effects of two or more factors. Statistical experimental design involves varying the factors together, the principal classical designs being various forms of factorial design. SA based on varying the inputs together rather than individually is called multi-way SA.

Through techniques analogous to the method of analysis of variance in Statistics, multi-way sensitivity analysis can identify interaction effects.

Regression SA

Thorough multi-way SA typically demands a large and highly structured set of simulator runs, even for quite modest numbers of inputs. In the same way as the analysis of variance is a particular case of regression analysis in Statistics, an alternative to multi-way SA is based on regression. Analysis involves fitting regression models to the outputs of available simulator runs.

If the regression model is a simple linear regression, the fitted slope parameters for the various inputs represent measures of sensitivity. However, such an approach cannot identify interactions, and shares most of the drawbacks of one-way SA. More thorough analysis will fit product terms for interactions, and potentially also nonlinear terms.

Probabilistic SA

Careful use and interpretation of multi-way or regression SA methods can yield quite comprehensive analysis of the relationship between the simulator's output and its inputs, for the purposes of understanding and/or dimension reduction. However, probabilistic SA was developed specifically to address the use of SA in the context of uncertain inputs. As remarked above, it was in response to uncertainty about inputs that SA evolved, but all of the preceding methods treat the uncertainty in the inputs only implicitly. In probabilistic SA the input uncertainty is explicit and described in a probability distribution $\omega(x)$.

Probabilistic SA is also a comprehensive approach to SA that can address interactions and nonlinearities, and it is preferred in the *MUCM* toolkit because it uniquely has the ability to characterise the relationship between input uncertainty and output uncertainty. It also extends naturally to address SA for decision-making.

8.115.3 Additional comments

Screening methods

Simulators often have very large numbers of inputs. Carrying out sophisticated SA methods on such a simulator can be highly demanding computationally. Only a small number of the inputs will generally have appreciable impact on the output, particularly when we are interested in the simulator's behaviour over a relatively small part of the possible input space. In practice, simple *screening* techniques are widely used initially to eliminate many inputs that have essentially no effect. Once this kind of drastic dimension reduction has been carried out, more formal and demanding SA techniques can be used to confirm and further refine the choice of *active inputs*. For a discussion of screening methods see the topic thread on screening (*ThreadTopicScreening*).

Range of variation

When we consider sensitivity of the output(s) to variations in an input, it can be important to define how (and in particular how far) that input might be varied. If we allow larger variations in x_j then we will often (although not necessarily) see larger impact on $f(x)$. In all of the approaches to SA discussed above, this issue arises. It is most obvious in one-way SA, where we have to specify the particular alternative value(s) for x_j, and clearly if we change those we may change the sensitivity and influence measures. It is also obvious in probabilistic SA, where the probability distribution assigned to x identifies in detail how the inputs are to be considered to vary.

As we have seen, the question of how (far) we allow x_j to vary does not obviously seem relevant at all for local SA, but the fact that the derivatives are not invariant to scale changes is an analgous issue. The case of regression SA is similar, because a coefficient in a simple linear regression fit is a gradient in the same way as a derivative and, in particular, changing the units of measurement affects such a coefficient in the same way. There is another aspect to the question for regression SA, though. If we fit a simple linear regression when $f(x)$ genuinely responds linearly to varying x_j, then the fitted gradient coefficient will not depend (or only minimally) on the range of variation we specify for x_j, but if the underlying response of $f(x)$ is non-linear then the coefficient in a linear fit will change if the range of variation changes.

In general, measures of sensitivity depend on how we consider perturbing the inputs. Changing the range of variation will change the SA measures and can alter the ranking of inputs according to which the outputs are most sensitive to. Probabilistic SA is no exception, but its emphasis on the careful specification of a probability distribution $\omega(x)$ over the input space avoids the often arbitrary way in which variations are defined in other SA approaches.

8.116 Alternatives: Prior specification for BL hyperparameters

8.116.1 Overview

In the fully *Bayes linear* approach to emulating a complex *simulator*, the *emulator* is formulated to represent prior knowledge of the simulator in terms of a *second-order belief specification*. The BL prior specification requires the specification of beliefs about some *hyperparameters*, as discussed in the alternatives page on emulator prior mean function (*AltMeanFunction*), the discussion page on the GP covariance function (*DiscCovarianceFunction*) and the alternatives page on emulator prior correlation function (*AltCorrelationFunction*). Specifically, in the *core problem* that is the subject of the core threads (*ThreadCoreBL*, *ThreadCoreGP*) a vector β defines the detailed form of the mean function, a scalar σ^2 quantifies the uncertainty or variability of the simulator around the prior mean function, while δ is a vector of hyperparameters defining details of the correlation function. Threads that deal with variations on the basic core problem may introduce further hyperparameters.

A Bayes linear analysis requires hyperparameters to be given prior expectations, variances and covariances. We consider here ways to specify these prior beliefs for the hyperparameters of the core problem. Prior specifications for other hyperparameters are addressed in the relevant variant thread. Hyperparameters may be handled differently in the fully *Bayesian* approach - see *ThreadCoreGP*.

8.116.2 Choosing the Alternatives

The prior beliefs should be chosen to represent whatever prior knowledge the analyst has about the hyperparameters. However, the prior distributions will be updated with the information from a set of training runs, and if there is substantial information in the training data about one or more of the hyperparameters then the prior information about those hyperparameters may be irrelevant.

In general, a Bayes linear specification requires statements of second-order beliefs for all uncertain quantities. In the current version of this Toolkit, the Bayes linear emulation approach does not consider the situation where σ^2 and δ are uncertain, and so we require the following:

- E$[\beta_i]$, Var$[\beta_i]$, Cov$[\beta_i, \beta_j]$ - expectations, variances and covariances for each coefficient β_i, and covariances between every pair of coefficients $(\beta_i, \beta_j), i \neq j$
- $\sigma^2 =$ Var$[w(x)]$ - the variance of the residual stochastic process
- δ - a value for the hyperparameters of the correlation function

8.116.3 The Nature of the Alternatives

Priors for β

Given a specified form for the basis functions $h(x)$ of $m(x)$ as described in the alternatives page on basis functions for the emulator mean (*AltBasisFunctions*), we must specify expectation and variance for each coefficient β_i and a covariance between every pair (β_i, β_j).

As with the basis functions $h(x)$, there are two primary means of obtaining a belief specification for β.

1. **Expert-led specification** - the specification can be made directly by an expert using methods such as

a. Intuitive understanding of the magnitude and impact of the physical effects represented by $h(x)$ leading to a direct quantification of expectations, variances and covariances.

b. Assessing the difference between the model under study and another well-understood model such as a fast approximate version or an earlier version of the same simulator. In this approach, we can combine the known information about the mean behaviour of the second simulator with the belief statements about the differences between the two simulator to construct an appropriate belief specification for the hyperparameters – see *multilevel emulation*.

2. **Data-driven specification** - when prior beliefs are weak and we have ample model evaluations, then prior values for β are typically not required and we can replace adjusted values for β with empirical estimates, $\hat{\beta}$, obtained by fitting the linear regression $f(x) = h(x)^T \beta$. Our uncertainty statements about β can then be deduced from the "estimation error" associated with $\hat{\beta}$.

Priors for σ^2

The current version of the Toolkit requires a point value for the variance about the emulator mean, σ^2. This corresponds directly to making a specification about Var$[w(x)]$. As with the model coefficients above, there are two possible approaches to making such a quantification. An expert could make the specification by directly quantifying the magnitude of σ^2. Alternatively, an expert assessment of the expected prior adequacy of the mean function at representing the variation in the simulator outputs can be combined with information on the variation of the simulator output, which allows for the deduction of a value of σ^2. In the case of a data-driven assessment, the estimate for the residual variance $\hat{\sigma}^2$ can be used.

In subsequent versions of the toolkit, Bayes linear methods will be developed for *learning* about σ^2 in the emulation process. This will require making prior specifications about the squared emulator residuals.

Priors for δ

Specification of correlation function hyperparameters is a more challenging task. Direct elicitation can be difficult as the hyperparameter δ is hard to conceptualise - the alternatives page on prior distributions for GP hyperparameters (*AltGPPriors*) provides some discussion on this topic, with particular application to the Gaussian correlation function. Alternatively, when given a large collection of simulator runs then δ can be crudely estimated using methods such as *variogram* fitting on the empirical residuals.

Assessing and updating uncertainties about δ raises both conceptual and technical problems as methods which would be optimal for assessing such parameters given realisations drawn from a corresponding stochastic process may prove to be highly non-robust when applied to functional computer output which is only represented very approximately by such a process. Methods for approaching this problem will appear in a subsequent version of the toolkit.

8.117 Alternatives: Basis functions for the emulator mean

8.117.1 Overview

The process of building an *emulator* of a *simulator* involves first specifying prior beliefs about the simulator and then updating this using a *training sample* of simulator runs. Prior specification may be either using the fully *Bayesian* approach in the form of a *Gaussian process* or using the *Bayes linear* approach in the form of first and second order moments. The basics of building an emulator using these two approaches are set out in the two core threads (*ThreadCoreGP*, *ThreadCoreBL*).

In either approach it is necessary to specify a mean function and covariance function. Choice of the form of the emulator prior mean function is addressed in *AltMeanFunction*. We consider here the additional problem of specifying the forms of the *basis functions* $h(x)$ when the form of the mean function is linear.

8.117.2 The Nature of the Alternatives

In order to specify an appropriate form of the global trend component, we typically first identify a set of potential basis functions appropriate to the simulator in question, and then select which elements of that set would best describe the simulator's mean behaviour via the methods described above. There are a huge number of possible choices for the specific forms of $h(x)$. Common choices include:

- Monomials - to capture simple large-scale effects
- Orthogonal polynomials - to exploit computational simplifications in subsequent calculations
- Fourier/trigonometric functions - to adequately represent periodic output
- A very fast approximate version of the simulator - to capture the gross simulator behaviour using an existing model

8.117.3 Choosing the Alternatives

When the mean function takes a linear form, an appropriate choice of the basis functions $h(x)$ is required. This is particularly true for a Bayes linear emulator as the emphasis is placed on a detailed structural representation of the simulator's mean behaviour.

Typically, there are two primary methods for determining an appropriate collection of trend basis functions:

1. prior information about the model can be used directly to specify an appropriate form for the mean function;
2. if the number of available model evaluations is very large, then we can empirically determine the form of the mean function from this data alone.

Expert information about $h(x)$ can be derived from a variety of sources including, but not limited to, the following:

- knowledge and experience with the computer simulator and its outputs;
- beliefs about the behaviour of the actual physical system that the computer model simulates;
- experience with similar computer models such as previous versions of the same simulator or alternative models for the same system;
- series expansions of the generating equations underlying the computer model (or an appropriately simplified model form);
- fast approximate versions of the computer model derived from simplifications to the current simulator.

If the prior information is weak relative to the available number of model evaluations and the computer model is inexpensive to evaluate, then we may choose instead to determine the form of the trend directly from the model evaluations. This empirical approach is reasonable since any Bayesian posterior would be dominated by the large volume of data. Thus in such situations it is reasonable to apply standard statistical modelling techniques. Empirical construction of the emulator mean is a similar problem to traditional regression model selection. In this case, methods such as stepwise model selection could be applied given a set of potential trend basis functions. However, using empirical methods to identify the form of the emulator mean function requires many more model evaluations than are required to fit an emulator with known form and hence is only applicable if a substantial number of simulator runs is available. Empirical construction of emulators using cheap simulators is a key component of *multilevel emulation* which will be described in a subsequent version of the Toolkit.

Often, the majority of the global variation of the output from a computer simulator $f(x)$ can be attributed to a relatively small subset, x_A, of the input quantities called the *active inputs*. In such cases, the emulator mean is considered to be a function of only the active inputs combined with a modified form of the covariance. Using this active input approach can make substantial computational savings; see the discussion page on active and inactive inputs (*DiscActiveInputs*) for further details. If the simulator has a high-dimensional input space then the elicitation of information about potential active inputs possibly after a suitable transformation of the input space, then and the form of at least some of the model effects can be very helpful in emulator construction (see Craig *et al.* 1998).

8.117.4 References

Craig, P. S., Goldstein, M., Seheult, A. H., and Smith, J. A. (1998) “Constructing partial prior specifications for models of complex physical systems,” *Applied Statistics*, **47**:1, 37–53

8.118 Alternatives: Training Sample Design for the Core Problem

8.118.1 Overview

An important step in the development of an *emulator* is the creation of a *training sample* of runs of the *simulator*. This entails running the simulator at each of a set of input points in order to see what output(s) the simulator produces at each of those input configurations. The set of input points is called the *design* for the training sample.

The design of training samples for building emulators falls within the wide field of statistical experimental design. The topic thread on experimental design (*ThreadTopicExperimentalDesign*) contains much more detail and discussion about design issues in *MUCM*, and on their relationship to conventional statistical experimental designs. Even within the realm of the MUCM toolkit there are many contexts in which designs are needed, each with its own criteria and yielding different kinds of solutions. We consider here the choice of design for a training sample to build an emulator for the “core problem.” The core problem is discussed in page *DiscCore*, and this page falls within the two core threads, the thread for the analysis of the core model using Gaussian process methods (*ThreadCoreGP*) and the thread for the Bayes linear emulation for the core model (*ThreadCoreBL*) - see the discussion of threads in the toolkit structure page (*MetaToolkitStructure*).

8.118.2 Choosing the Alternatives

The training sample design needs to facilitate the development of an emulator that correctly and accurately predicts the actual simulator output over a desired region of the input space. By convention, we suppose that the input region of interest, $\mathcal{X}$, is the unit cube in p dimensions, $[0, 1]^p$. We assume that this has been achieved by transforming each of the p simulator inputs individually by a simple linear transformation, and that the i-th input corresponds to the i-th dimension of $\mathcal{X}$. (More extended discussion on technical issues in training sample design for the core problem can be found in page *DiscCoreDesign*.)

There are several criteria for a good design, which are also discussed in *DiscCoreDesign*, but the most widely adopted approach in practice is to employ a design in which the points are spread more or less evenly and as far apart as possible within $\mathcal{X}$. Such a design is called-space-filling. The detailed alternatives that are discussed below all have space-filling properties.

8.118.3 The Nature of the Alternatives

Factorial design

Factorial design over $\mathcal{X}$ defines a set of points L_j in $[0, 1]$ called levels for dimension $j = 1, 2, \ldots, p$. Then if the number of points in L_j is n_j the full factorial design takes all $n = \prod_{j=1}^{p} n_j$ combinations of the levels to produce a p-dimensional grid of points in $\mathcal{X}$. If the levels are spread out over $[0, 1]$ in each dimension (so as to have a space-filling property in each dimension), then the full factorial is a space-filling design. However, if p is reasonably large then even when the number of levels takes the minimal value of two in each dimension the size $n = 2^p$ of the design becomes prohibitively large for practical simulators.

Fractional factorial designs consist of subsets of the full factorial that generally have some pattern or symmetry properties, but although some such designs can be realistically small it is usual to adopt alternative ways to construct space-filling designs.

Optimised Latin hypercube designs

A Latin hypercube (LHC) is a set of points chosen randomly subject to a constraint that ensures that the values of each input separately are spread evenly across $[0, 1]$. The procedure for constructing a LHC is given in *ProcLHC*. LHCs are not guaranteed to be space-filling in $\mathcal{X}$, just in each dimension separately. It is therefore usual to generate a large number of random LHCs and to select from these the one that best satisfies a specified criterion.

One popular criterion is the minimum distance between any two points in the design. Choosing the LHC with the maximal value of this criterion helps to ensure that the design is well spread out over $\mathcal{X}$, and a LHC optimised according to this criterion is known as a maximin LHC design. This and other criteria are discussed in *DiscCoreDesign*.

The procedure for generating an optimised LHC, according to any desired criterion and in particular according to the maximin criterion, is given in the procedure for generating an optimised Latin hypercube design (*ProcOptimalLHC*).

Non-random space-filling design

A number of different sequences of numbers have been proposed that have space-filling properties. The can be thought of as pseudo-random sequences. The sequences use different algorithms to generate them, but all have the property that they are potentially infinite in length, and a design of n points is obtained simply by taking the first n points in the sequence.

- Lattice designs. A lattice is a special grid of n points in $[0, 1]^d$. It is defined by d generators, and each successive point is obtained by adding a constant (depending on the generator) to each coordinate and then reducing back to $[0, 1]$. If the generators are well-chosen the result can be a good space-filling design. The procedure for generating a lattice design, with suggestions on choice of generators, is given in the procedure for generating a lattice design (*ProcLatticeDesign*).
- Weyl sequences. A Weyl sequence is similar to a lattice design in the way it is generated, but with generators that are irrational numbers. See the procedure for generating a Weyl design (*ProcWeylDesign*).
- Halton sequences. A Halton sequence also has a prime integer "generator" for each dimension, and each prime generates a sequence of fractions. For instance, the generator 2 produces the sequence $\frac{1}{2}, \frac{1}{4}, \frac{3}{4}, \frac{1}{8}, \frac{5}{8}, \frac{3}{8}, \frac{7}{8}, \frac{1}{16}, \ldots$. So if the i-th dimension has generator 2 then these will be the i-th coordinates of successive points in the Halton sequence. See the procedure for generating a Halton design (*ProcHaltonDesign*).
- Sobol's sequence. The Sobol's sequence uses the same set of coordinates as a Halton sequence with generator 2 for each dimension, but then reorders them according to a complicated rule. If we used the Halton sequence in $p = 2$ dimensions with generator 2 for both dimensions, we would get the sequence $(\frac{1}{2}, \frac{1}{2}), (\frac{1}{4}, \frac{1}{4}), (\frac{3}{4}, \frac{3}{4}), \ldots$, and so on, so that all the points would lie on the diagonal of $[0, 1]^2$. The Sobol's sequence reorders the coordinates of each successive block of 2^i points $(i = 0, 1, 2, \ldots)$ in a LHC way. For instance, the Sobol's sequence for $p = 2$ begins $(\frac{1}{2}, \frac{1}{2}), (\frac{1}{4}, \frac{3}{4}), (\frac{3}{4}, \frac{1}{4}), \ldots$. The complexity of the algorithm is such that we do not provide an explicit procedure in the *MUCM* toolkit, but we are aware of two freely available algorithms (*disclaimer*). For users of the R programming language, we suggest the function `runif.sobol(n,d)` from the package `fOptions`` in the R repository. The Sobol's sequence is sometimes known also as the LP-tau sequence, and the GEM-SA software package also generates Sobol's designs under this name. For more explanation and insight into the Sobol's sequence, see the Sobol's sequence procedure page (*ProcSobolSequence*).

Model based optimal design

Optimal design seeks a design which maximises/minimises some function, typically, of the covariance matrix of the parameters or predictions. Different *optimality criteria* can be chosen for the classical optimal design. Formal optimisation may lead to space-filling designs but may also yield designs which are better tailored to specific emulation requirements. There is more information about model based optimal design in *ThreadTopicExperimentalDesign*. In particular, MUCM is developing a sequential strategy to select the design, called ASCM (*Adaptive Sampler for Complex Models*), that will eventually make use of the *Karhunen Loeve expansion* to approximate the Gaussian process.

8.118.4 Additional Comments, References, and Links

Because the optimised LHC designs require very many random LHCs to be generated in order to choose the best one, this kind of design takes substantially longer to generate than the non-random sequences. They also have the disadvantage that the procedure is random and so repeating it to generate a new design with the same number of points and dimensions will produce a different result.

Another advantage of the Weyl, Halton and Sobol's sequences is that we can readily add further points to the design. This facilitates the idea of sequential design, where the training set is steadily increased until a sufficiently good and validated emulator is obtained.

On the other hand, the non-random designs can be difficult to tune to get good space-filling properties; only the Sobol's sequence does not require careful choice of a set of generators. These designs can also produce clumps or ridges of points.

The Halton and Sobol's sequences are examples of low-discrepancy sequences. Discrepancy is a measure of departure of a set of points from a uniform spread over $[0, 1]^p$ and these are some of a small number of sequence generators that have been shown to have asymptotically minimal discrepancy. For more details, and in particular for a full description of the Sobol's sequence, see

- Kuipers, L. and Niederreiter, H. (2005). Uniform distribution of sequences. Dover Publications, ISBN 0-486-45019-8

Finally, the optimised LHC designs have additional flexibility through the optimality criterion that may allow them to adapt better to prior information about the simulator; see *DiscCoreDesign*.

8.119 Alternatives: Emulator prior correlation function

8.119.1 Overview

The process of building an *emulator* of a *simulator* involves first specifying prior beliefs about the simulator and then updating this using a *training sample* of simulator runs. Prior specification may be either using the fully *Bayesian* approach in the form of a *Gaussian process* (GP) or using the *Bayes linear* approach in the form of first and second order moments. The basics of building an emulator using these two approaches are set out in the two core threads: the thread for the analysis of the core model using Gaussian process methods (*ThreadCoreGP*) and the thread for the Bayes linear emulation for the core model (*ThreadCoreBL*).

In either approach it is necessary to specify a covariance function. The formulation of a covariance function is considered in the discussion page on the GP covariance function (*DiscCovarianceFunction*). Within the *MUCM* toolkit, the covariance function is generally assumed to have the form of a variance (or covariance matrix) multiplied by a correlation function. We present here some alternative forms for the correlation function $c(\cdot, \cdot)$, dependent on hyperparameters δ.

8.119.2 Choosing the Alternatives

The correlation function $c(x, x')$ expresses the correlation between the simulator outputs at input configurations x and x', and represents the extent to which we believe the outputs at those two points should be similar. In practice, it is formulated to express the idea that we believe the simulator output to be a relatively smooth and continuous function of its inputs. Formally, this means the correlation will be high between points that are close together in the input space, but low between points that are far apart. The various correlation functions that we will consider in this discussion of alternatives all have this property.

8.119.3 The Nature of the Alternatives

The Gaussian form

It is common, and convenient in terms of subsequent building and use of the emulator, to specify a covariance function of the form

$$c(x,x') = \exp\left[-0.5(x-x')^T C(x-x')\right]$$

where C is a diagonal matrix whose diagonal elements are the inverse squares of the elements of the δ vector. Hence, if there are p inputs and the i-th elements of the input vectors x and x' and the *correlation length* vector δ are respectively x_i, x'_i and δ_i, we can write

$$\begin{aligned} c(x,x') &= \exp\left\{-\sum_{i=1}^{p} 0.5\left[(x_i - x'_i)/\delta_i\right]^2\right\} \\ &= \prod_{i=1}^{p} \exp\left\{-0.5\left[(x_i - x'_i)/\delta_i\right]^2\right\}. \end{aligned}$$

This formula shows the role of the correlation length hyperparameter δ_i. The smaller its value, the closer together x_i and x'_i must be in order for the outputs at x and x' to be highly correlated. Large/small values of δ_i therefore mean that the output values are correlated over a wide/narrow range of the i-th input x_i. See the discussion of "smoothness" at the end of this page.

Because of its similarity to the density function of a normal distribution, this is known as the Gaussian form of correlation function.

A generalised Gaussian form

The second expression in equation (2) has the notable feature that the correlation function across the whole x space is a product of correlation functions referring to each input separately. A correlation function of this form is said to be *separable*. A generalisation of the Gaussian form that does not entail separability is

$$c(x,x') = \exp\left[-0.5(x-x')^T M(x-x')\right]$$

where now M is a symmetric matrix with elements in the vector δ. So if there are p inputs the δ vector has $p(p+1)/2$ elements, whereas in the simple Gaussian form it has only p elements. The hyperparameters are now also more difficult to interpret.

In practice, this generalised form has rarely been considered. Its many extra hyperparameters are difficult to estimate and the greater generality seems to confer little advantage in terms of obtaining good emulation.

The exponential power form

An alternative generalisation replaces (2) by

$$c(x,x') = \prod_{i=1}^{p} \exp\left[-\{|x_i - x'_i|/\delta_{1i}\}^{\delta_{2i}}\right]$$

where now in addition to correlation length parameters δ_{1i} we have power parameters δ_{2i}. Hence δ has $2p$ elements. This is called the exponential power form and has been widely used in practice because it allows the expression of alternative kinds of *regularity* in the simulator output. The simple Gaussian form has $\delta_{2i} = 2$ for every input, but values less than 2 are also possible. The value of 2 implies that as input i is varied the output will behave very regularly, in the sense that the simulator output will be differentiable with respect to x_i. In fact it implies that the output will be differentiable with respect to input i infinitely many times.

If $1 < \delta_{2i} < 2$ then the output will be differentiable once with respect to x_i but not twice, while if the value is less than or equal to 1 the output will not be differentiable at all (but will still be continuous).

We would rarely wish to allow the power parameters to get as low as 1, since it is hard to imagine any simulator whose output is not differentiable with respect to one of its inputs at *any* point in the parameter space. However, it is hard to distinguish between a function that is once differentiable and one that is infinitely differentiable, and allowing power parameters between 1 and 2 can give appreciable improvements in emulator fit.

Matérn forms

Another correlation function that is widely used in some applications is the Matérn form, which for a one-dimensional x is

$$c(x,x') = \frac{2^{1-\delta_2}}{\Gamma(\delta_2)} \left(\frac{x-x'}{\delta_1}\right)^{\delta_2} \mathcal{K}_{\delta_2}\left(\frac{x-x'}{\delta_1}\right)$$

where $\mathcal{K}_{\delta_2}(\cdot)$ is a modified Bessel function of the third kind, δ_1 is a correlation length parameter and δ_2 behaves like the power parameter in the exponential power family, controlling in particular the existence of derivatives of the simulator. (The number of derivatives is δ_2 rounded up to the next integer.)

There are natural generalisations of this form to x having more than one dimension.

Adding a nugget

The Gaussian form with nugget modifies the simple Gaussian form (1) to

$$c(x,x') = \nu I_{x=x'} + \exp\left[-0.5(x-x')^T C(x-x')\right]$$

where the expression $I_{x=x'}$ is 1 if $x = x'$ and is otherwise zero, and where ν is a *nugget* term. A nugget can similarly be added to any other form of correlation function.

There are three main reasons for adding a nugget term in the correlation function.

Nugget for computation

The simple Gaussian form (1) and the generalised Gaussian form (3) allow some steps in the construction and use of emulators to be simplified, with resulting computational benefits. However, the high degree of regularity (see discussion below) that they imply can lead to computational problems, too.

One device that is sometimes used to address those problems is to add a nugget. In this case, ν is not usually treated as a hyperparameter to be estimated but is instead set at a small fixed value. The idea is that this small modification is used to achieve computational stability (in situations that will be set out in the relevant pages of this toolkit) and ideally ν should be as small as possible.

Technically, the addition of a nugget implies that the output is now not even continuous anywhere, but as already emphasised this is simply a computational device. If the nugget is small enough it should have negligible effect on the resulting emulator.

Nugget for inactive inputs

When some of the available inputs of the simulator are treated as *inactive* and are to be ignored in building the emulator, then a nugget term may be added to represent the unmodelled effects of the inactive inputs; see also the discussion on active and inactive inputs (*DiscActiveInputs*).

In this case, ν would normally be treated as an unknown hyperparameter, and so is added to the set δ of hyperparameters in the correlation function. The nugget's magnitude will depend on how much of the output's variation is due to the inactive inputs. By the nature of inactive inputs, this should be relatively small but its value will generally need to be estimated as a hyperparameter.

Nugget for stochastic simulators

The use of a nugget term to represent the randomness in outputs from a *stochastic* simulator is similar to the way it is introduced for inactive inputs. A thread dealing with stochastic simulators will be incorporated in a future release of the MUCM toolkit.

Other forms of correlation function

In principle, one could consider a huge variety of other forms for the correlation function, and some of these might be useful for special circumstances. However, the above cases represent all those forms that have been used commonly in practice. In the machine learning community several other flexible correlation functions have been defined which are claimed to have a range of desirable properties, in particular non-stationary behaviour. A particular example which has not yet been applied in emulation is the so called "neural-network correlation function" which was developed by Chris Williams and can be thought of as representing a sum of an infinite hidden layer multilayer perceptron, described in Gaussian Processes for Machine Learning.

8.119.4 Additional Comments, References, and Links

The notion of regularity is important in constructing a suitable correlation function. Regularity is defined in the *MUCM* toolkit as concerning continuity and differentiability, and in particular the more derivatives the simulator has, the more regular it is. The Gaussian form always has infintely many derivatives, and so expresses a strong belief in the simulator output responding to its inputs in a very regular way. The Matérn form, on the other hand, allows any finite positive number of derivatives, depending on the δ_2 hyperparameter. The exponential power form allows for the simulator output to be infinitely differentiable, not differentiable anywhere, or just once differentiable.

A function that is at least once differentiable is in practice almost indistinguishable from one that is infinitely differentiable, so the extra flexibility of the Matérn form may not be a material advantage. Whenever we can be confident that the output is continuous and differentiable the Gaussian form could be used, and this confers computational benefits in the creation and use of emulators. When we suspect that the output could respond more irregularly, so that it is not differentiable everywhere, then the exponential power form is recommended.

A related property is that of *smoothness*, which in the MUCM toolkit concerns how rapidly the simulator output can "wiggle" as we vary the inputs. This is in practice controlled by correlation length parameters, which are often referred to as smoothness or roughness parameters. The higher the values of these hyperparameters, the less likely the output is to "wiggle" over any given range of inputs.

In all this discussion of correlation functions, it is important to remember that a discontinuous mean function will cause the output to be discontinuous regardless of any regularity specified for the correlation function, and a wiggly mean function will cause the output to wiggle regardless of any smoothness properties of the correlation function. We must therefore think of the correlation function as describing the behaviour of the simulator output *after* the mean function is subtracted.

As already discussed, the form of the correlation function specifies how smooth we expect the simulator output to be as the inputs are varied, with the hyperparameters δ being estimated from the training data to identify the correlation function fully. A fully Bayesian analysis will require prior distributions to be specified for the hyperparameters, whereas slightly different procedures apply in a Bayes linear analysis. This step is addressed in the appropriate thread, e.g. *ThreadCoreGP* or *ThreadCoreBL*.

8.120 Alternatives: Dynamic Emulation Approaches

8.120.1 Overview

This page discusses two approaches for building an *emulator* of a *dynamic simulator*. Dynamic simulators produce multivariate outputs: a time series of *state variables* $w_1, \dots, w_T$. The strategy for emulating such a simulator presented in the variant thread for dynamic emulation (*ThreadVariantDynamic*) is to construct an emulator of the *single step function* $w_t = f(w_{t-1}, a_t, \phi)$. An alternative is to treat the simulator like any other multivariate output simulator, and construct an emulator directly as in the variant thread for the analysis of a simulator with multiple outputs (*ThreadVariantMultipleOutputs*). Here, we consider the relative merits of the two approaches.

8.120.2 The Nature of the Alternatives

1. Build an emulator of the single step function $w_t = f(w_{t-1}, a_t, \phi)$. If we wish to obtain a joint distribution of $w_1, \dots, w_T$ given $a_1, \dots, a_T$ and ϕ, we cannot do so directly. We must either use simulation methods (*ProcExactIterateSingleStepEmulator*) or an approximation approach (*ProcApproximateIterateSingleStepEmulator*).
2. Build an emulator of the full times series simulator $(w_1, \dots, w_T) = f_{full}(w_0, a_1, \dots, a_T, \phi)$. This simulator and corresponding emulator have a much larger input and output space, but the emulator will directly give us the joint distribution of $w_1, \dots, w_T$ given $w_0, a_1, \dots, a_T$ and ϕ. We may only be interested in some subvector or function of $(w_1, \dots, w_T)$, which could be emulated directly, reducing the dimension of the output space.

8.120.3 Choosing the Alternatives

We argue that emulating the single step function is preferable under any of the following circumstances:

1. We wish to emulate the outputs over a range of different series of *forcing inputs* $a_1, \dots, a_T$. This may be because we are uncertain about the value of a ‘true’ forcing input series, or we may just wish to investigate how the outputs vary as $a_1, \dots, a_T$ varies. In a multivariate output emulator, the inputs are $w_0, \phi, a_1, \dots, a_T$. Hence the input dimension is large if T is large, and building any emulator becomes increasingly difficult as the number of inputs increases. In the single step emulator, the inputs are w_{t-1}, ϕ, a_t, and so the number of emulator inputs is fixed regardless of the value of T.
2. We are uncertain about the maximum T of interest. A ‘black box’ emulator of the function $(w_1 \dots, w_T) = f_{full}(w_0, \phi, a_1, \dots, a_T)$ cannot predict w_{T+t} for any $t > 0$. It is, however, possible to extrapolate if
   - the forcing variables are fixed and are not treated as emulator inputs
   - t is treated as an additional simulator/emulator input
   - The prior mean function is a good approximation of the relationship between w_t and t in the simulator. See the alternatives page on emulator prior mean function (*AltMeanFunction*)
3. We wish to increase the complexity of the simulator single step function. As the computational expense of the single step function increases, it may become increasingly impractical to obtain sufficient training runs of the full times series simulator, depending on the value of T.

8.120.4 Additional Comments, References and Links

When emulating the full time series output directly, there are particular emulator modelling choices that can speed up computation considerably. These are presented in

Rougier, J. C. (2008), Efficient Emulators for Multivariate Deterministic Functions, *Journal of Computational and Graphical Statistics*, 17(4), 827-843.

8.121 Alternatives: Estimators of correlation hyperparameters

8.121.1 Overview

As discussed in the procedure for building a Gaussian process emulator for the core problem (*ProcBuildCoreGP*), one approach to dealing with uncertainty in the *hyperparameters* δ of the correlation function in a *Gaussian process emulator* is to ignore it, using a single estimate of δ instead of a sample from its posterior distribution. We discuss here a number of alternative ways of obtaining a suitable estimate.

8.121.2 Choosing the Alternatives

The chosen value should be a representative central value in the posterior distribution $\pi^*(\cdot)$. Where the parameters have the interpretation of correlation lengths, as is the case in most of the correlation functions discussed in the alternatives page for the emulator prior correlation function (*AltCorrelationFunction*), the estimate should preferably err on the side of under-estimating these parameters rather than over-estimating them. This will lead to an emulator with a little more uncertainty in its predictions, which may compensate somewhat for not formally accounting for uncertainty in δ.

8.121.3 The Nature of the Alternatives

The most widely used estimator, as mentioned in *ProcBuildCoreGP*, is the posterior mode. This is the value of δ at which $\pi^*(\delta)$ is maximised. Although any convenient algorithm or utility might be used for this purpose, care should be taken because it is not uncommon to find multiple local maxima in $\pi^*(\cdot)$. If several modes are found, the choice between them might, for instance, be made using the cross-validation approach mentioned below, or by seeing which choice leads to an emulator with better *validation* diagnostics (see the procedure page for validating a Gaussian process emulator (*ProcValidateCoreGP*)).

A related estimator is $\exp(\bar{\lambda})$, where $\bar{\lambda}$ is defined in the procedure for a multivariate lognormal approximation for correlation hyperparameters (*ProcApproxDeltaPosterior*) as the mode of $\ln \delta$. It should be noted, though, that this will tend to produce larger values of correlation length parameters than the simple mode of δ.

Another method that has been suggested, and is widely used in spatial statistics, although there is little experience about how it will perform in the emulator setting, is cross-validation. The idea here is to compare alternative candidate values of δ by how well they predict the training data outputs when the emulator is built from a subset of the training data. In its simplest form, the cross-validation measure for a given candidate value of δ is computed as follows.

1. For $j = 1, 2, \ldots, n$ compute the posterior mean $\hat{f}_j = m^*(x_j)$ of the j-th training sample output using (a) the formula for the posterior mean given in *ProcBuildCoreGP*, (b) the candidate value of δ and (c) the $n-1$ training sample runs obtained by leaving out the j-th run.
2. Then the cross-validation measure is $\sum_{j=1}^{n}(f(x_j) - \hat{f}_j)^2$.

Small values of the cross-validation measure are best. In the full cross-validation method, all possible values of δ are candidates, with an algorithm being applied to search for the value minimising the cross-validation measure.

Another approach that is very widely used in spatial statistics is variogram fitting, as described in the procedure for variogram estimation of covariance function hyperparameters (*ProcVariogram*). Although this is advocated in the *Bayes linear* approach (see the thread for the Bayes linear emulation for the core model (*ThreadCoreBL*)), in the fully *Bayesian* approach using a *Gaussian process* emulator it is not recommended because it involves inappropriate approximations/simplifications.

Experience in spatial statistics may be unreliable for our purposes because that experience is typically limited to two or three dimensions, equivalent in MUCM terms to emulating simulators with only two or three inputs.

8.121.4 Additional Comments, References, and Links

Further research is being conducted in *MUCM* into this question, and findings will be reported in this page in due course.

8.122 Alternatives: Prior distributions for GP hyperparameters

8.122.1 Overview

In the fully *Bayesian* approach to *emulating* a complex *simulator*, a *Gaussian process* (GP) is formulated to represent prior knowledge of the simulator. The GP specification is conditional on some *hyperparameters*, as discussed in the alternatives pages for emulator prior mean function (*AltMeanFunction*), and correlation function (*AltCorrelationFunction*) and the discussion page on the GP covariance function (*DiscCovarianceFunction*). Specifically, in the *core problem* that is the subject of the core threads using Gaussian process methods (*ThreadCoreGP*) or Bayes linear (*ThreadCoreBL*) a vector β defines the detailed form of the mean function, a scalar σ^2 quantifies the uncertainty or variability of the simulator around the prior mean function, while δ is a vector of hyperparameters defining details of the correlation function. Threads that deal with variations on the basic core problem may introduce further hyperparameters.

In particular in the multi output variant (see the thread on the analysis of a simulator with multiple outputs using Gaussian process methods (*ThreadVariantMultipleOutputs*)) there are a range of possible parameterisations which require different (more general) prior specifications. The univariate case is typically the simplification to scalar settings. For example if a input-output *separable* multi output emulator is used, then the prior specification will be over a matrix β, and a matrix Σ and a set of correlation function hyperparameters δ. Priors for this case are discussed in the alternatives page on prior distributions for multivariate GP hyperparameters (*AltMultivariateGPPriors*).

A fully Bayesian analysis requires hyperparameters to be given prior distributions. We consider here alternative ways to specify prior distributions for the hyperparameters of the core problem. Prior distributions for other hyperparameters are addressed in the relevant variant thread. Hyperparameters may be handled differently in the *Bayes Linear* approach - see *ThreadCoreBL*.

8.122.2 Choosing the Alternatives

The prior distributions should be chosen to represent whatever prior knowledge the analyst has about the hyperparameters. However, the prior distributions will be updated with the information from a set of training runs, and if there is substantial information in the training data about one or more of the hyperparameters then the prior information about those hyperparameters may be essentially irrelevant.

In general we require a *joint* distribution $\pi(\beta, \sigma^2, \delta)$ for all the hyperparamters. Where required, we will denote the marginal distribution of δ by $\pi_\delta(\delta)$, and similarly for marginal distributions of other groups of hyperparameters.

8.122.3 The Nature of the Alternatives

Priors for σ^2

In most applications, there will be plenty of information about σ^2 in the training data. We typically have at least 100 training runs, and 100 observations would usually be considered adequate to estimate a variance. Unless there is strong prior information available regarding this hyperparameter, it would be acceptable to use the conventional *weak prior* specification

$$\pi_{\sigma^2}(\sigma^2) \propto \sigma^{-2}$$

independently of the other hyperparameters.

In situations where the training data are more sparse, which may arise for instance when the simulator is computationally demanding, prior information about σ^2 may make an important contribution to the analysis. Genuine prior information about σ^2 in the form of a *proper* prior distribution should be specified by a process of *elicitation* - see references at the end of this page. See also the discussion of conjugate prior distributions below.

Priors for β

Again, we would expect to find that in most applications there is enough evidence in the training data to identify β well, particularly when the mean function is specified in the linear form, so that the elements of β are regression parameters. Then it is acceptable to use the conventional weak prior specification

$$\pi_{\beta}(\beta) \propto 1$$

independently of the other hyperparameters.

If there is a wish to express genuine prior information about β in the form of a proper prior distribution, then this should be specified by a process of elicitation - see references at the end of this page. See also the discussion of conjugate prior distributions below.

Conjugate priors for β and σ^2

When substantive prior information exists and is to be specified for either β or σ^2, then it is convenient to use *conjugate* prior distributions if feasible.

If prior information is to be specified for σ^2 alone (with the weak prior specification adopted for β), the conjugate prior family is the inverse gamma family. This can be elicited using the SHELF package referred to at the end of this page.

If β is a vector of regression parameters in a linear form of mean function, and prior information is to be specified about both β and σ^2, then the conjugate prior family is the normal inverse gamma family. Specifying such a distribution is a complex business - see the reference to Oakley (2002) at the end of this page.

Although these conjugate prior specifications make subsequent updating using the training data as simple as in the case of weak priors, the details are not given in the *MUCM* toolkit because it is expected that weak priors for β and σ^2 will generally be used. If needed, information on using conjugate priors in building an emulator can be found in the Oakley (2002) reference at the end of this page. (The case of an inverse gamma prior for σ^2 combined with a weak prior for β is a special case of the general normal inverse gamma prior.)

If prior information is to be specified for β alone, the conjugate prior family is the normal family, but for full conjugacy the variance of β should be proportional to σ^2 in the same way as is found in the normal inverse gamma family. This seems unrealistic when weak prior information is to be specified for σ^2, and so we do not discuss this conjugate option further.

Priors for δ

The situation with δ is quite different, in that the information in the training data will often fail to identify these hyperparameters to sufficient accuracy. It can therefore be difficult to estimate their values effectively, and inappropriate estimates can lead to poor emulation. Also, it is known that in the case of a Gaussian or an exponential power correlation function (see the alternatives page on emulator prior correlation function (*AltCorrelationFunction*)) conventional weak prior distributions may lead to improper posterior distributions. See the alternatives page on estimators of correlation hyperparameters (*AltEstimateDelta*).

Accordingly, prior information about δ can be very useful, or even essential, in obtaining a valid emulator. Fortunately, genuine prior information about δ often exists.

Consider the case of a Gaussian correlation function, where the elements of δ are correlation lengths which define the *smoothness* of the simulator output as each input is varied. The experience of the users and developers of the simulator may suggest how stable the output should be in response to varying individual inputs. In particular, it may be possible to specify a range for each input such that it is not expected the output will "wiggle" (relative to the mean function). For other forms of correlation function, there may again be genuine prior information about the smoothness and/or *regularity* that is controlled by various elements of δ. We suggest that even quite crudely elicited distributions for these parameters will be better than adopting any default priors.

8.122.4 Additional Comments, References, and Links

The following resources on elicitation of prior distributions are mentioned in the text above. The first is a thorough review of the field of elicitation, and provides a wealth of general background information on ideas and methods. The second (SHELF) is a package of documents and simple software that is designed to help those with less experience of elicitation to elicit expert knowledge effectively. SHELF is based on the authors' own experiences and represents current best practice in the field. Finally, the third reference deals specifically with eliciting a conjugate prior for β and σ^2 in a GP model with a linear mean function.

O'Hagan, A., Buck, C. E., Daneshkhah, A., Eiser, J. R., Garthwaite, P. H., Jenkinson, D. J., Oakley, J. E. and Rakow, T. (2006). Uncertain Judgements: Eliciting Expert Probabilities. John Wiley and Sons, Chichester. 328pp. ISBN 0-470-02999-4.

SHELF - the Sheffield Elicitation Framework - can be downloaded from http://tonyohagan.co.uk/shelf (*Disclaimer*)

Oakley, J. (2002). Eliciting Gaussian process priors for complex computer codes. *The Statistician* 51, 81-97.

The following paper discusses issues of propriety in GP posterior distributions related to the choice of prior when it is desired to express weak prior information (about δ as well as other hyperparameters).

Paulo, R. (2005). Default priors for Gaussian processes. *Annals of Statistics*, 33, 556-582.

8.123 Alternatives: Gaussian Process or Bayes Linear Emulators

8.123.1 Overview

The *MUCM* technology of quantifying and managing uncertainty in complex simulation models rests on the idea of *emulation*. This toolkit deals with two basic ways to construct and use emulators - *Gaussian process* emulators and *Bayes linear* emulators. The two forms have much in common, but there are also fundamental differences. These differences have impact on the ways in which the two approaches are used and the kinds of applications to which they are applied.

8.123.2 Choosing the Alternatives

- To choose the route of Gaussian process emulation, the core thread is *ThreadCoreGP*.
- To choose the route of Bayes linear emulation, the core thread is *ThreadCoreBL*.
- The core threads deal with emulation of a basic kind of problem called the *core problem*. For more complex situations, the relevant variant thread deals with how to modify the core to tackle new features.

8.123.3 The Nature of the Alternatives

An emulator is a statistical representation of a *simulator*. For any given values of the simulator's inputs, we can obtain the simulator output(s) by running the simulator itself, but we can instead use an emulator to predict what the output(s) would be. MUCM methods use the emulator to address questions about the simulator far more efficiently than methods which rely on running the simulator itself.

Perhaps the most fundamental difference between GP and BL emulators is the nature of the predictions.

- A GP emulator provides a full probability distribution for the output(s) as its prediction. In particular, the mean of the distribution is the natural estimate, and the variance (or its square root, the standard deviation) provides a measure of accuracy. Because it provides a full predictive distribution, a GP emulator can also provide credible intervals for the outputs.
- A BL emulator provides an estimate and a dispersion measure that are analogous to the mean and variance of a GP emulator, although in principle they have somewhat different interpretations. BL emulator predictions, however, are not full probability distributions, and are confined to the estimate and dispersion measure.

These differences result from a difference in underlying philosophy. GP emulators are based in conventional *Bayesian* statistical theory, in which data are represented by their sampling distributions and unknown parameters by prior or posterior distributions. Bayes linear theory is based on considering a set of uncertain quantities and defining a belief specification for them which comprises an estimate and a measure of dispersion for each, together with an association measure for each pair. From the perspective of conventional Bayesian statistics, it is natural to interpret these as the means, variances and covariances of the uncertain quantities, and in some ways this interpretation conforms with how these measures are elicited and used in Bayes linear methods. However, the Bayes linear view is philosophically different, and in its most abstract form interprets them geometrically - the estimates define a point in an abstract metric space with a metric defined by the dispersion and association measures. The Bayes linear analogue of the use of Bayes' theorem in Bayesian theory, to update beliefs in the light of additional information, is projection in the metric space.

The two theories coincide if in the Bayes linear analysis the set of uncertain quantities include the indicator functions of every possible value for all the random variables that are given probability distributions in the fully Bayesian approach. However, this equivalence is achieved by the Bayes linear analyst effectively defining full probability distributions, and in practice Bayes linear analyses will only exceptionally include such a full probabilistic specification. Adherents of the Bayes linear view argue, correctly, that one can never make so many judgements about a problem if those judgements are to represent carefully considered beliefs. In a fully Bayesian analysis, distributions are not specified by thinking about every single probability that makes up that distribution. In reality, they are specified by a convenient (and more loosely considered) completion of a much smaller number of carefully considered judgements. Proponents of the Bayes linear approach decline in principle to make judgements that are not individually considered and elicited.

The choice between the two approaches can then be reduced to choosing between two alternative ways of making statistical inferences in the context of a relatively small, finite number of actual, carefully considered judgements.

1. In the full Bayesian approach, those judgements are expanded to produce full probability distributions, by a choice of distributional form that is partly based on informal judgement but also partly on convenience. Then updating via Bayes' theorem produces full posterior distributions for inference.

2. In the Bayes linear approach, a complete Bayes linear belief specification is required for the chosen set of uncertain quantities, but no other judgements are added (and in particular distributions are not required). Updating is via projection and yields an adjusted belief specification.

The two approaches differ in their interpretation of the meaning of the belief analysis, but each approach may be regarded, from the other viewpoint, as an approximation to what would be produced using their preferred approach.

It is not the purpose of this page to go deeply into the philosophy. Although Bayes linear methods have only limited support in the Bayesian community generally, they are accepted in the field of computer model uncertainty, and have made important contributions to that field.

The choice between a GP and a BL emulator is principally a matter of philosophical viewpoint but also partly pragmatic. The computations required for BL methods are based on manipulating variance matrices, and can remain tractable even in very complex applications, where sometimes the computations using the GP emulator become infeasible. On the other hand, it may be difficult to formulate the complete belief specification in a complex BL application, and not having full posterior distributions makes it more difficult to deliver some kinds of inferences in the BL approach.

8.124 Alternatives: Implausibility Measures

8.124.1 Description and Background

As introduced in *ThreadGenericHistoryMatching*, an integral part of the *history matching* process is the use of *implausibility measures*. This page discusses implausibility measures in more detail and defines several different types, highlighting their strengths and weaknesses. The notation used is the same as that defined in *ThreadGenericHistoryMatching*. Here we use the term model synonymously with the term *simulator*.

8.124.2 Discussion

Univariate Implausibility Measures

Our approach to history matching is based on the assessment of certain implausibility measures which we now describe. An implausibility measure is a function defined over the input space which, when large, suggests that the match between model and system would exceed a stated tolerance in terms of the relevant uncertainties that are present, such as the *model discrepancy* d and the observational errors e; see *ThreadVariantModelDiscrepancy* for more details. We may build up this concept of an acceptable match for a single output $f_i(x)$ as follows. For a given choice of input x, which we now consider as a candidate for the *best input* x^+, we would ideally like to assess whether the output $f_i(x)$ differs from the system value y_i by more than the tolerance that we allow in terms of model discrepancy. Therefore, we would assess the standardised distance

$$\frac{(y_i - f_i(x))^2}{\mathrm{Var}[d_i]}$$

In practice, we cannot observe the system y_i directly and so we must compare $f_i(x)$ with the observation z_i, introducing measurement error e_i with corresponding standardised distance

$$\frac{(z_i - f_i(x))^2}{\mathrm{Var}[d_i] + \mathrm{Var}[e_i]}$$

However, for most values of x, we are not able to evaluate $f(x)$ as the model takes a significant time to run, so instead we use the emulator and compare z_i with $\mathrm{E}[f_i(x)]$. Therefore, the implausibility function or measure is defined as

$$I^2_{(i)}(x) = \frac{(\mathrm{E}[f_i(x)] - z_i)^2}{\mathrm{Var}[\mathrm{E}[f_i(x)] - z_i]} = \frac{(\mathrm{E}[f_i(x)] - z_i)^2}{\mathrm{Var}[f_i(x)] + \mathrm{Var}[d_i] + \mathrm{Var}[e_i]}$$

where the second equality results from the fact that, here, x is implicitly being considered as a candidate for the best input x^+, and hence the independence assumptions that feature in the *best input* approach (see *DiscBestInput*) can be used. Note that we could have a more complex structure describing the link between model and reality, in which case the denominator of the implausibility measure would be altered. When $I_{(i)}(x)$ is large, this suggests that, even given all the uncertainties present in the problem, namely, the emulator uncertainty $\mathrm{Var}[f(x)]$, the model discrepancy $\mathrm{Var}[d]$ and the observational errors $\mathrm{Var}[e]$, we would be unlikely to view as acceptable the match between model output and observed data were we to run the model at input x. Therefore, we consider that choices of x for which $I_{(i)}(x)$ is large can be discarded as potential members of the set $\mathcal{X}$ of all acceptable inputs. We discard regions of the input space by imposing suitable cutoffs on the implausibility measure as is discussed in *DiscImplausibilityCutoff*.

Note that this definition of the univariate implausibility is natural and comes from the concept of standardised distances. As we will simply be imposing cutoffs on such measures, any results will be invariant to a monotonic transformation of the implausibility measure.

Multivariate Implausibility Measures

So far we have a separate implausibility measure $I_{(i)}(x)$ for each of the outputs labelled by i that were considered for the history matching process. Note that not all outputs have to be used: we can select a subset of the outputs that are deemed to be representative in some sense; e.g., using principal variables (Cumming, 2007). We may now choose to make some intuitive combination of the individual implausibility measures as a basis of eliminating portions of the input space, or we may construct the natural multivariate analogue.

A common such combination is the maximum implausibility measure $I_M(x)$ defined as

$$I_M(x) = \max_i I_{(i)}(x)$$

Discarding every x such that $I_M(x) > c$ is equivalent to applying the cutoff to every individual $I_{(i)}(x)$. While this is a simple, intuitive and commonly used measure, it is sensitive to problems caused by inaccurate emulators: if one of the emulators is performing poorly at input x, this could lead to x being wrongly rejected. For this reason the second and the third maximum implausibility measures $I_{2M}(x)$ and $I_{3M}(x)$ are often used (Vernon 2010, Bower 2009), defined using set notation as:

$$I_{2M}(x) = \max_i \left(I_{(i)}(x) \setminus I_M(x) \right)$$

$$I_{3M}(x) = \max_i \left(I_{(i)}(x) \setminus I_M(x), I_{2M}(x) \right)$$

that is we define $I_{2M}(x)$ and $I_{3M}(x)$ to be the second and third highest value out of the set of univariate measures $I_i(x)$ respectively. These measures are much less sensitive to the failings of individual emulators.

We can construct the full multivariate implausibility measure $I_{MV}(x)$ provided we have suitable multivariate expressions for the model discrepancy d and the observational errors e (see *ThreadVariantModelDiscrepancy*). Normally a full multivariate emulator is also required, however a multi-output emulator of sufficient accuracy can also be used (see *ThreadVariantMultipleOutputs*).

$I_{MV}(x)$ takes the form:

$$I_{MV}(x) = (z - \mathrm{E}[f(x)])^T (\mathrm{Var}[z - \mathrm{E}[f(x)]])^{-1} (z - \mathrm{E}[f(x)])$$

which becomes

$$I_{MV}(x) = (z - \mathrm{E}[f(x)])^T (\mathrm{Var}[f(x)] + \mathrm{Var}[d] + \mathrm{Var}[e])^{-1} (z - \mathrm{E}[f(x)])$$

Here, $\mathrm{Var}[f(x)]$, $\mathrm{Var}[d]$ and $\mathrm{Var}[e]$ are all $r \times r$ covariance matrices and z and $\mathrm{E}[f(x)]$ are r-vectors, where r is the number of outputs chosen for use in the history matching process.

The multivariate form is more effective for screening the input space, but it does require careful consideration of the covariance structure for the various quantities involved, especially the model discrepancy d.

The history matching process requires that we choose appropriate cutoffs for each of the above measures: possible choices of such cutoffs are discussed in *DiscImplausibilityCutoff*.

8.124.3 Additional Comments

The implausibility measures based on summaries of univariate measures, such as $I_M(x)$ and $I_{2M}(x)$, tend to select inputs that correspond to runs where all or most of the outputs are within a fixed distance from each of the observed data points. While this is useful, if the outputs represent points on some physical function, this type of implausibility measure will not capture the *shape* of this function. The fully multivariate measure $I_{MV}(x)$ on the other hand, can be constructed to favour runs which mimic the correct shape of the physical function. The extent of this effect depends critically on the structure of the model discrepancy and observational error covariance matrices.

8.124.4 References

Cumming, J. A. and Wooff, D. A. (2007), "Dimension reduction via principal variables," *Computational Statistics & Data Analsysis*, 52, 550–565.

Vernon, I., Goldstein, M., and Bower, R. (2010), "Galaxy Formation: a Bayesian Uncertainty Analysis," MUCM Technical Report 10/03.

Bower, R., Vernon, I., Goldstein, M., et al. (2009), "The Parameter Space of Galaxy Formation," to appear in MNRAS; MUCM Technical Report 10/02.

8.125 Alternatives: Iterating single step emulators

8.125.1 Overview

This page is concerned with task of *emulating* a *dynamic simulator*, as set out in the variant thread for dynamic emulation (*ThreadVariantDynamic*).

We have an emulator for the *single step function* $w_t = f(w_{t-1}, a_t, \phi)$ in a dynamic simulator, and wish to iterate the emulator to obtain the distribution of $w_1, \ldots, w_T$ given $w_0, a_1, \ldots, a_T$ and ϕ. For a *Gaussian Process* emulator, it is not possible to do so analytically, and so we consider two alternatives: a simulation based approach and an approximation based on the normal distribution.

8.125.2 The Nature of the Alternatives

1. Recursively simulate random outputs from the single step emulator, adding the simulated outputs to the training data at each iteration to obtain an exact draw from the distribution of $w_1, \ldots, w_T$ (*ProcExactIterateSingleStepEmulator*).
2. Approximate the distribution of $w_t = f(w_{t-1}, a_t, \phi)$ at each time step with a normal distribution. The mean and variance of w_t for each t are computed recursively. (*ProcApproximateIterateSingleStepEmulator*).

8.125.3 Choosing the Alternatives

The simulation method can be computationally intensive, but has the advantage of producing exact draws from the distribution of $w_1, \dots, w_T$. The approximation method is computationally faster to implement, and so is to be preferred if the normal approximation is sufficiently accurate. The approximation is likely to be accurate when

- uncertainty about $f(\cdot)$ is small, and
- the simulator is approximately linear over any small part of the input space.

As always, it is important to validate the emulator (see the procedure page on validating a GP emulator (*ProcValidateCoreGP*)), but particular attention should be paid to emulator uncertainty in addition to emulator predictions.

8.125.4 Additional comments

If the approximation method is to be used for many different choices $w_0, a_1, \dots, a_T$ and ϕ, it is recommended that both methods are tried for a subset of these choices, to test the accuracy of the approximation.

8.126 Alternatives: Emulator prior mean function

8.126.1 Overview

The process of building an *emulator* of a *simulator* involves first specifying prior beliefs about the simulator and then updating this using a *training sample* of simulator runs. Prior specification may be either using the fully *Bayesian* approach in the form of a *Gaussian process* or using the *Bayes linear* approach in the form of first and second order moments. The basics of building an emulator using these two approaches are set out in the two core threads: the thread for the analysis of core model using Gaussian process methods (*ThreadCoreGP*) and the thread for the Bayes linear emulation for the core model (*ThreadCoreBL*).

In either approach it is necessary to specify a mean function and covariance function. We consider here the various alternative forms of mean function that are dealt with in the *MUCM* toolkit. An extension to the case of a vector mean function as required by the thread for the analysis of a simulator with multiple outputs using Gaussian methods (*ThreadVariantMultipleOutputs*) can be found in a companion page to this one, dealing with alternatives for multi-output mean functions (*AltMeanFunctionMultivariate*).

8.126.2 Choosing the Alternatives

The mean function gives the prior expectation for the simulator output at any given set of input values. We assume here that only one output is of interest, as in the *core problem*.

In general, the mean function will be specified in a form that depends on a number of *hyperparameters*. Thus, if the vector of hyperparameters for the mean function is β then we denote the mean function by $m(\cdot)$, so that $m(x)$ is the prior expectation of the simulator output for vector x of input values.

In principle, this should entail the analyst thinking about what simulator output would be expected for every separate possible input vector x. In practice, of course, this is not possible. Instead, $m(\cdot)$ represents the general shape of how the analyst expects the simulator output to respond to changes in the inputs. The use of the unknown hyperparameters allows the emulator to learn their values from the training sample data. So the key task in specifying the mean function is to think generally about how the output will respond to the inputs.

Having specified $m(\cdot)$, the subsequent steps involved in building and using the emulator are described in *ThreadCoreGP* / *ThreadCoreBL*.

8.126.3 The Nature of the Alternatives

The linear form

It is usual, and convenient in terms of subsequent building and use of the emulator, to specify a mean function of the form:

$$m(x) = \beta^T h(x)$$

where $h(\cdot)$ is a vector of (known) functions of x, known as *basis functions*. This is called the linear form of mean function because it corresponds to the general linear regression model in statistical analysis. When the mean function is specified to have the linear form, it becomes possible to carry out subsequent analyses more simply. The number of elements of the vector $h(\cdot)$ will be denoted by q. These elementary functions are called *basis functions*.

There remains the choice of $h(\cdot)$. We illustrate the flexibility of the linear form first through some simple cases.

- The simplest case is when $q = 1$ and $h(x) = 1$ for all x. Then the mean function is $m(x) = \beta$, where now β is a scalar hyperparameter representing an unknown overall mean for the simulator output. This choice expresses no prior knowledge about how the output will respond to variation in the inputs.
- Another simple instance is when $h(x)^T = (1, x)$, so that $q = 1 + p$, where p is the number of inputs. Then $m(x) = \beta_1 + \beta_2 x_1 + \ldots + \beta_{1+p} x_p$, which expresses a prior expectation that the simulator output will show a trend in response to each of the inputs, but there is no prior information to suggest any specific nonlinearity in those trends.
- Where there is prior belief in nonlinearity of response, then quadratic or higher polynomial terms might be introduced into $h(\cdot)$.

In principle, all of the kinds of linear regression models that are used by statisticians are available for expressing prior expectations about the simulator. Some further discussion of the choice of basis functions is given in the alternatives page for basis functions for the emulator mean (*AltBasisFunctions*) and the discussion page on the use of a structured mean function (*DiscStructuredMeanFunction*).

Other forms of mean function

Where prior information suggests that the simulator will respond to variation in its inputs in ways that are not captured by a regression form, then it is possible to specify any other mean function.

For example,

$$m(x) = \beta_1/(1 + \beta_2 x_1) + \exp{(\beta_3 x_2)}$$

expresses a belief that as the first input, x_1 increases the simulator output will flatten out in the way specified in the first term, while as x_2 increases the output will increase (or decrease if $\beta_3 < 0$) exponentially. Such a mean function might be used where the prior information about the simulator is suitably strong, but this cannot be cast as a regression form. As a result, the analysis (as required for building the emulator and using it for tasks such as *uncertainty analysis*) will become more complex.

Mean functions appropriate for the multivariate output setting are discussed in *AltMeanFunctionMultivariate*.

8.126.4 Additional Comments, References, and Links

It is important to recognise that the emulator specification does not say that the emulator will respond to its inputs in exactly the way expressed in the mean function. The Gaussian process, or its Bayes linear analogue, will allow the actual simulator output to take any form at all, and given enough training data will adapt to the true form regardless of what is specified in the prior mean. However, the emulator will perform better the more accurately the mean function reflects the actual behaviour of the simulator.

As already discussed, the form of the mean function specifies the shape that we expect the output to follow as the inputs are varied, with the hyperparameters β being estimated from the training data to identify the mean function fully. A fully Bayesian analysis will require a prior distribution to be specified for β, while a Bayes linear analysis will require a slightly different form of prior information. This step is addressed in the appropriate core thread, *ThreadCoreGP* or *ThreadCoreBL*.

8.127 Alternatives: Multivariate emulator prior mean function

8.127.1 Overview

In the process of building a *multivariate GP* emulator it is necessary to specify a mean function for the GP model. The mean function for a set of outputs is a vector function. Its elements are the mean functions for each output, defining a prior expectation of the form of that output's response to the simulator's inputs. Alternative choices on the emulator prior mean function for an individual output are discussed in *AltMeanFunction*. Here we discuss how these may be adapted for use in the multivariate emulator.

8.127.2 Choosing the alternatives

If the linear form of mean function is chosen with the same :math:qtimes 1 vector $h(\cdot)$ of *basis functions<DefBasisFunctions>* for each output, then the vector mean function can be expressed simply as $\beta^T h(\cdot)$, where β is a $q \times r$ matrix of regression coefficients and r is the number of outputs. The choice of regressors for the two simple cases shown in *AltMeanFunction* can then be adapted as follows:

* For $q = 1$ and $h(x) = 1$, the mean function becomes $m(x) =$
$beta$, where β is a r-dimension vector of hyperparameters representing an unknown multivariate overall mean (i.e. the mean vector) for the simulator output.

* For $h(x)^T = (1, x)$, so that $q = 1 + p$, where p is the number of inputs and the mean function is an r-dimensional vector whose j{m(x)}_j=beta_{1,j} + \beta_{2,j},x_1 + \ldots + \beta_{1+p,j},x_p‘.

This linear form of mean function in which all outputs have the same set of basis functions is generally used in the multivariate emulator described in *ThreadVariantMultipleOutputs*, where it is the natural choice for the *multivariate Gaussian process* and has mathematical and computational advantages. However, the linear form does not have to be used, and there may be situations in which we believe the outputs behave like a function that is non-linear in the unknown coefficients, such as $\{m(x)\}_j = x/(\beta_{1,j} +$
$beta_{2,j}x)$. The general theory still holds for a non-linear form, albeit without the mathematical simplification which allow the analytic integration of the regression parameters β.

It would also be possible to use different mean functions for different outputs, but this is not widely done in practice and renders the notation more verbose.

8.128 Alternatives: Approaches to emulating multiple outputs

8.128.1 Overview

The simplest analyses in the *MUCM* toolkit relate to the *core problem*. One of the simplifying restrictions in the core problem is that we are only interested in one output from the *simulator*. In the core threads that use either Gaussian process methods (*ThreadCoreGP*) or Bayes linear methods (*ThreadCoreBL*), the core problem is addressed using the MUCM methodology of first building an *emulator* and then answering relevant questions about the simulator output by means of the emulator. In reality, simulators almost always produce multiple outputs, and we are often interested in addressing questions concerning more than one output.

One of the most common tasks is to predict multiple outputs of the simulator at input points where the output has not yet been observed. In a sense, this can be considered as a question of predicting each output separately, reducing the problem to several individual core problems. However, prediction inevitably involves uncertainty and in predicting several outputs we are concerned about joint uncertainty, represented by covariances as well as variances. If we only predict each output separately by the core problem methods of *ThreadCoreGP* or *ThreadCoreBL*, we are ignoring covariances between the outputs. This could cause significant over-estimation of our uncertainty, and the problem will become more serious as the correlation between outputs increases. If the outputs are uncorrelated (or independent) then using separate emulators will give good results.

A related task is to predict one or more functions of the outputs. For instance, a simulator may have outputs that represent amounts of rainfall at a number of locations, but we wish to predict the total rainfall over a region, which is the sum of the rainfall outputs at the various locations (or perhaps a weighted sum if the locations represent subregions of different sizes). Or if the simulator outputs the probability of a natural disaster and its consequence in loss of lives, then the product of these two outputs is the expected loss of life, which may be of primary interest. For questions involving functions of outputs, in the fully Bayesian approach we can use a simulation based inference procedure, as detailed in *ProcPredictMultiOutputFunction*. (This is not generally an option in the Bayes linear approach.) For linear functions, it may be possible to provide specific methods for prediction and related tasks.

In general, we can tackle questions of multiple outputs using the same basic MUCM methodology as for a single output (i.e. we emulate the various outputs and then use the resulting emulator(s) to address the relevant tasks) but there are now a number of alternative ways in which we might emulate multiple simulator outputs.

Consistency

A consideration that might be important in the choice of alternative is consistency. For instance, if we emulate multiple outputs together by one of these methods, would the predictions for any single output be the same as we would have produced if we had just emulated that one output as in *ThreadCoreGP* or *ThreadCoreBL*?

More generally, if we build an emulator for multiple outputs and use it to predict one or more functions of the outputs, would this give the same predictions as if we had taken the output vectors from our training sample, computed the relevant function(s) for each output vector and then fitted an emulator to these? For instance, if we take the training sample outputs and sum the values of the first two outputs, then fit a single-output emulator to those sums, would this give the same result as fitting an emulator for the multiple outputs and then making predictions about the sum of the first two outputs?

We cannot hope to achieve consistency of this kind in all cases. For instance, we should not expect to have consistency for predicting nonlinear functions of outputs. A Gaussian process emulator for the product of two outputs, for example, will not look like the product of two Gaussian processes. Some kinds of multiple output emulators, though, will give consistency for prediction either of individual outputs or of linear functions of outputs.

8.128.2 Choosing the Alternatives

We present here a number of alternative approaches to emulation of multiple outputs. In each case we consider the complexity of building the relevant emulator(s) and of using them to address tasks such as prediction, uncertainty analysis or sensitivity analysis of some combination of outputs.

It is worth noting that whatever approach we choose for emulation the training data will comprise a set of output vectors $f(x_i), i = 1, 2, \ldots, n$. Each output vector contains values for all the outputs of interest.

Independent outputs

A simple approach is to assume that the outputs are independent (or in the Bayes linear framework, just uncorrelated). Then we can build separate, independent emulators for the different outputs using the methods set out in the core threads *ThreadCoreGP* and *ThreadCoreBL*. This allows different mean and correlation functions to be fitted for each output, and in particular different outputs can have different correlation length parameters in their hyperparameter vectors δ. Therefore this approach does not assume that the input and output correlations are *separable*.

Using the independent emulators for tasks such as prediction, uncertainty and sensitivity analyses is set out in *ThreadGenericMultipleEmulators*, and is also relatively simple.

The independent outputs approach can also been seen as a special case of the general multivariate emulator (see below and *ThreadVariantMultipleOutputs*), in which the multivariate covariance function is diagonal (see *AltMultivariateCovarianceStructures*).

A hybrid approach of breaking the outputs into groups such that there is near independence between groups but we expect correlation within groups might be considered. Then we emulate each group separately using one of the methods discussed below. It is not difficult to adapt the procedures to accommodate this case, but we do not provide explicit procedures in this version of the toolkit.

Transformation of outputs to independence

Another approach is to transform the outputs so that the resulting transformed outputs can be considered to be independent (or uncorrelated), and can then be emulated using independent emulators as discussed above. If such a transformation can be found, then we can proceed as above. Since the original outputs are functions of the transformed outputs, we can address any required task, such as prediction or sensitivity analysis, using the same methods as described in the generic thread for combining two or more emulators (*ThreadGenericMultipleEmulators*).

The only additional questions to address with this approach are (a) how to find a suitable transformation and (b) how to allow for uncertainty in the choice of transformation.

We will refer to the transformed outputs which will be emulated independently as latent outputs, to distinguish these from the original outputs of the simulator.

Transformation methods

The simplest transformations to consider are linear. In principle, we could use nonlinear transformations but we do not consider that possibility here.

One approach is to use the method of principal components, or some other variance decomposition method, as described in *ProcOutputsPrincipalComponents*. These methods in general produce a one-to-one linear transformation, with the same number of latent outputs as original outputs of interest. Thus, we can represent the latent outputs as a linear transformation of original outputs, use this transformation to compute the latent output values for each output vector in the training sample and proceed to develop independent single-output emulators for the latent outputs. The original outputs are then derived using the inverse linear transformation and we can apply the methods of *ThreadGenericMultipleEmulators* for prediction, uncertainty or sensitivity analysis.

We can also consider linear transformations in which there are fewer latent outputs than original outputs. This possibility arises naturally when we use principal components, since the essence of that method is to find a set of latent outputs such that as much as possible of the variability in the original outputs is explained by the first few latent outputs. If the remaining latent outputs are very insensitive to variation in the simulator inputs (over the range explored in the training sample), then we may consider that there is little value in emulating these. In practice there will be no guarantee that lower variance components might be less structured and some sort of method for looking at correlation structure, for example variograms and cross-variograms for the latent outputs could be used to provide a qualitative assessment of the stability of this model.

Formally, we represent the latent outputs that we do not emulate fully as zero mean GPs with a correlation function that consists only of a nugget (setting :math:nu=1 in *AltCorrelationFunction<AltCorrelationFunction>*). The variance hyperparameter σ^2 is equated in each case to the estimated variance of that principal component. We can then apply the methods of *ThreadGenericMultipleEmulators* for prediction, etc.

This kind of dimension reduction can be a very powerful way of managing the emulation of large numbers of outputs, and remains the only widely explored method for tackling high dimensional outputs. This form of dimension reduction is likely to be effective when there might be expected to be strong correlation between the outputs of the simulator, as for example is often the case where the output of the simulator is composed of a grid (or other tessellation / projection) of a spatial or spatio-temporal field. Indeed, there is often much redundancy in outputs on a grid since in order for the simulator to resolve a spatial feature effectively it must be larger than around 5-10 grid lengths (depending on the simulator details), otherwise it can induce numerical instabilities in the simulator, or be poorly modelled.

> Uncertainty in transformation choice

When we have applied one of the above methods to find a suitable transformation, we may wish to acknowledge formally that the transformation has only been estimated. We are supposing here that there is a 'true' transformation such that the latent outputs would be independent, but that we are unsure of what that transformation would be. We then need to treat the transformation as providing additional uncertain hyperparameters. For instance, if we are selecting a linear transformation, the coefficients of that transformation become the new hyperparameters. We can in principle represent this uncertainty through sample values of the transformation hyperparameters, with which we augment the sampled values of any other hyperparameters. Although in principle the necessary hyperparameter samples can be derived from the statistical procedure used to estimate the transformation, and this uncertainty can then be taken account of as described in *ThreadGenericMultipleEmulators*, this will not be simple in practice. Furthermore, the estimation of the transformation hyperparameters should be done jointly with the fitting of the emulator hyperparameters to avoid double use of the data. We do not offer detailed procedures for this in the current version of the toolkit, although we may do so in a future release.

The alternative is to ignore such additional uncertainty. We have selected a transformation without any particular belief that there is a 'true' transformation that would make the latent outputs independent. Rather, we think that this transformation will simply make the assumption of independence that we necessarily make when developing separate emulators a better representation of the simulator. Under this view questions of double use of the data are also ignored.

Outputs as extra input dimension(s)

In some simulators, the outputs represent values of some generic real-world property at different locations and/or times. For instance, a simulator of a vehicle crash may output velocities or forces on the vehicle as a time series at many time points during the simulated crash. As another example, a simulator of the spread of an oil-spill in the ocean may output the concentration of oil at different geographic locations in the ocean and at different times.

In such cases, an alternative way to treat the multiple outputs is as a single output but with one or more extra inputs.

- In the crash simulator, we add the time as an input. Then if we are interested in the velocity output we represent the simulator as producing a single velocity output when given all the original inputs plus a time input value. This is now a single output simulator and can be tackled simply by the methods of *ThreadCoreGP* or *ThreadCoreBL*.
- Similarly, if we consider the oil-spill simulator, we need to add three or four inputs - the time value and the geographic coordinates (either 2 of these or, if the simulation also covers depth within the ocean, 3). Then again we have a single output simulator and can use one of the core threads to emulate it.
- Note also that if we are interested in both velocity and force in the crash simulator, then adding a time input leaves a simulator with two outputs, velocity and force. We therefore still need to use methods for emulating multiple inputs, but the device of making time an input has reduced the number of outputs enormously (from twice the number of time points to just two).

This approach is clearly very simple, but we need also to consider what restrictions it implies for the covariance structure in terms of the original outputs. In theory, there need not be any restriction, since in principle the covariance function of the single-output emulator with the new inputs can be defined to represent any joint covariance structure. In practice, however, the covariance structures that we generally assume make quite strong restrictive assumptions. As discussed in *DiscCovarianceFunction* we invariably assume a constant variance, which implies that all of the original outputs have the same variance. This may be reasonable, but often is not - for instance, we may expect changes in the other simulator inputs to have increasing effects on oil-spill concentrations over time. Furthermore, the usual choices of correlation function considered in *AltCorrelationFunction* make strong assumptions. For instance, the Gaussian and power exponential correlation functions have the effect of (a) assuming separability between the original inputs and outputs, and (b) assuming a form of correlation function between the original outputs that is unlikely to be appropriate, particularly in the case of time series.

So the great simplicity of the additional input approach is balanced by restrictive assumptions that may lead to poor emulation (or may require much larger training samples to achieve adequate emulation).

The multivariate emulator

In order to relax some of the restrictions imposed by the additional input approach, we need to directly emulate all the outputs simultaneously using a multivariate emulator. The multivariate emulator is an extension of the *Gaussian process emulator for the core problem*, based on a *multivariate Gaussian process*. The key differences between the multivariate emulator and the additional input approach are (a) the multivariate emulator does not restrict the outputs to all having the same variance, and (b) the multivariate emulator does not impose a parametric form (e.g. Gaussian or power exponential) on the between-output correlations.

The methodology for building and using a multivariate emulator is presented in *ThreadVariantMultipleOutputs*. A key issue is in choosing the covariance function for the *multivariate Gaussian process*. The various multivariate covariance functions that are available are discussed in *AltMultivariateCovarianceStructures*.

The dynamic emulator

A quite different approach in the case of *dynamic* simulators which produce time series of outputs by recursively updating a state vector is to emulate the single step simulator. A separate thread, *ThreadVariantDynamic*, deals with dynamic emulation.

8.128.3 Additional Comments, References, and Links

The multivariate emulator with a separable covariance function is developed and contrasted with the approach of treating outputs as an extra input in

Conti, S. and O'Hagan, A. (2009). Bayesian emulation of complex multi-output and dynamic computer models. Journal of Statistical Planning and Inference. doi: 10.1016/j.jspi.2009.08.006

In the following paper, the authors develop a dynamic linear model representation for outputs that are in the form of a time series. A special case of this is equivalent to (a) formulating the outputs as a time input and (b) the multivariate emulator with a separable covariance function, but with a proper time series structure for the covariance between outputs. However, their model is made more general by allowing the time series model parameters to vary.

Liu, F, and West, M. (2009). A dynamic modelling strategy for Bayesian computer model emulation. Bayesian Analysis 4, 393-412. Published online at http://ba.stat.cmu.edu/journal/2009/vol04/issue02/liu.pdf.

8.129 Alternatives: Choice of covariance function in the multivariate Gaussian process emulator

8.129.1 Overview

The covariance function in the multivariate Gaussian process emulator for a simulator with r outputs is the $r \times r$ matrix function $v(.,.)$. The i, j represents the covariance between $f_i(x)$ and $f_j(x')$. That means that $v(.,.)$ must encode two kinds of correlation: correlation between the outputs, and correlation over the input space. In order to build these two kinds of correlation into a valid covariance function (that is, one for which the covariance matrix $v(D, D)$ is positive semi-definite for any design D), we must impose some structure on $v(.,.)$. Here we describe some alternatives for this structure.

Do covariances matter?

Suppose that we are able to emulate all the outputs of interest accurately using single-output emulators, so that the uncertainty in each case is very small. Then we will know that the true values of any two simulator outputs will be very close to the emulator mean values. Knowing also the covariance will allow us to assess just how probable it is that the true values for both outputs lie above their respective emulator means, but when the uncertainty is very small this extra information is of little importance.

So it may be argued that if we have enough training data to emulate all outputs of interest very accurately, in the sense that the emulation uncertainty is small enough for us to be able to ignore it, then we can also ignore covariances between outputs. It is generally the case that the more accurately we can emulate the outputs the less importance attaches to our choice of emulation strategy, and so with large training samples all alternatives considered here will give essentially the same results. However, the reality with complex simulators is that we are rarely in the happy position of being able to make enough training runs to achieve this kind of accuracy, particularly if the emulator input and output spaces are high dimensional. When emulator uncertainty is appreciable, covariances between outputs need to be considered and the choice between alternative ways of emulating multiple outputs does matter.

8.129.2 The Alternatives

Independent outputs

If we are willing to treat the outputs as being independent then the covariance function $v(.,.)$ is diagonal, with diagonal elements $\sigma_1^2 c_1(.,.), \ldots, \sigma_r^2 c_r(.,.)$, where each output i has its own input space correlation function $c_i(.,.)$. The hyperparameters in this case are $\omega = (\sigma^2, \delta)$, where $\sigma^2 = (\sigma_1^2, \ldots, \sigma_r^2)$ and $\delta = (\delta_1, \ldots, \delta_r)$, with δ_i being the correlation length hyperparameters for $c_i(.,.)$. A typical prior for $\omega = (\sigma^2, \delta)$ is of the form

$$\pi(\sigma^2, \delta) \propto \pi_\delta(\delta) \prod_{i=1}^{r} \sigma_i^{-2} \,,$$

which expresses weak knowledge about σ^2. Alternatives for the correlation length prior, $\pi_\delta(.)$, are discussed in *AltGPPriors*.

Separable covariance

A *separable* covariance has the form

$$v(.,.) = \Sigma c(\cdot, \cdot) \,,$$

where Σ is a $r \times r$ covariance matrix between outputs and $c(.,.)$ is a correlation function between input points. The hyperparameters for the separable covariance are $\omega = (\Sigma, \delta)$, where δ are the hyperparameters for $c(.,.)$. A typical prior for $\omega = (\Sigma, \delta)$ is of the form

$$\pi(\Sigma, \delta) \propto \pi(\delta)|\Sigma|^{\frac{k+1}{2}}\,,$$

which expresses weak knowledge about Σ. This is also discussed in *AltMultivariateGPPriors* where the variance parameters are considered together with those for the mean function and δ is assumed known or fixed. The choice of $\pi(\delta)$, the arbitrary prior on the correlations lengths, is discussed in *AltGPPriors*.

Nonseparable covariance based on convolution methods

One way of constructing a Gaussian process is by taking the convolution of a Gaussian white noise process with an arbitrary *smoothing kernel*. This method lends itself to defining a nonseparable multivariate covariance function, which is often called a 'convolution covariance function'.

To construct a convolution covariance function $v(.,.)$, we choose a smoothing kernel κ_i corresponding to each output i. We also have a positive semidefinite $r \times r$ matrix $\tilde{\Sigma}$ with elements $\tilde{\sigma}_{ij}$. Then for each $i, j \in \{1, ..., r\}$, the i, j is

$$v_{ij}(x, x') = \tilde{\sigma}_{ij} \int_{R^p} \kappa_i(u - x)\kappa_j(u - x')\, du\,.$$

The smoothing kernels determine the input space correlation function for each output, and $\tilde{\Sigma}$ controls the between-output covariances. The hyperparameters in the convolution covariance function are $\omega = (\tilde{\Sigma}, \delta)$, where $\delta = \{\delta_1, ..., \delta_r\}$ are the parameters in the smoothing kernels, which will typically correspond to something like correlation lengths.

A convenient choice of smoothing kernel is

$$\kappa_i(x) = \left[\left(\frac{4}{\pi}\right)^p \prod_{\ell=1}^{p} \phi_i^{(\ell)}\right]^{\frac{1}{4}} \exp\{-2x^T \Phi_i x\}\,,$$

where each Φ_i is a $p \times p$ diagonal matrix with diagonal elements $(\phi_i^{(1)}, ..., \phi_i^{(p)} = ((\delta_i^{(1)})^{-2}, ..., (\delta_i^{(p)})^{-2})$, for then the covariance function has elements

$$v_{ij}(x, x') = \tilde{\Sigma}_{ij}\rho_{ij} \exp\{-2(x - x')^T \Phi_i(\Phi_i + \Phi_j)^{-1}\Phi_j(x - x')\}\,,$$

where

$$\rho_{ij} = 2^{\frac{p}{2}} \prod_{\ell=1}^{p} \left[(\phi_i^{(\ell)}\phi_j^{(\ell)})^{\frac{1}{4}}(\phi_i^{(\ell)} + \phi_j^{(\ell)})^{-\frac{1}{2}}\right]\,.$$

This means that the covariance function for an individual output i is $v_{ii}(x, x') = \sigma_i^2 \exp\{-(x - x')^T \Phi_i(x - x')\}$. Thus each output i has an input space correlation function of the Gaussian (squared exponential) form, as described in *AltCorrelationFunction*, with correlation lengths $\delta_i = (\delta_i^{(1)}, \ldots, \delta_i^{(p)}) = ((\phi_i^{(1)})^{-1/2}, \ldots, (\phi_i^{(p)})^{-1/2})$. A possible choice of prior for the hyperparameters is

$$\pi(\tilde{\Sigma}, \delta) \propto \pi(\delta)|\tilde{\Sigma}|^{\frac{k+1}{2}}$$

where $\pi(\delta)$ is an arbitrary prior on the correlations lengths, as discussed in *AltGPPriors*. Note that the hyperparameter $\tilde{\Sigma}$ is not itself the between-outputs covariance matrix, so this prior is not directly equivalent to the weak prior for the separable covariance function given above.

Nonseparable covariance based on the linear model of coregionalization

The linear model of coregionalization (LMC) was developed in the field of geostatistics as a tool to model multivariate spatial processes, and provides an alternative way of constructing a nonseparable multivariate covariance function, which we call the 'LMC covariance function'.

To construct a LMC covariance function $v(.,.)$, we choose a set of r 'basis correlation functions' $\kappa_1(.,.),\dots,\kappa_r(.,.)$. These can be any correlation function, as discussed in *AltCorrelationFunction*. We also have the $r \times r$ between-outputs covariance matrix Σ, for which we must choose a square-root decomposition $\Sigma = RR^T$. We usually choose R to be the symmetric eigendecomposition of Σ, that is $R = Q\mathrm{diag}(\sqrt{d_1},\dots,\sqrt{d_r})Q^T$ where $d_1,\dots,d_r$ are the eigenvalues of Σ and Q is a matrix with the eigenvectors of Σ as its columns (arranged from left to right in the order corresponding to $d_1,\dots,d_r$). An advantage of using the symmetric eigendecompostion of Σ is that the resulting emulator is not dependent on the ordering of the outputs, but note that other square-root decompositions, such as the Cholesky decomposition, may be used.

For any given choice of decomposition $\Sigma = RR^T$, the LMC covariance function is

$$v(.,.) = \sum_{i=1}^{r} \Sigma_i \kappa_i(.,.)\,,$$

where, for each $i = 1,...,r$, $\Sigma_i = r_i r_i^T$, with r_i the i.

We can see that the covariance function for an individual output is a weighted sum of the basis correlation functions. The weights are determined by the elements of the 'coregionalization matrices' $\{\Sigma_i : i = 1,...,k\}$. With the LMC covariance function, the emulator covariance matrices defined in *ProcBuildMultiOutputGP* become sums of Kronecker products of the coregionalization matrices and basis correlation matrices:

$$V = \sum_{i=1}^{r} \Sigma_i \otimes \kappa_i(D,D)\,,$$

$$u(x) = \sum_{i=1}^{r} \Sigma_i \otimes \kappa_i(D,x)\,.$$

The hyperparameters in the LMC covariance function are $\omega = (\Sigma, \tilde{\delta})$, where $\tilde{\delta}$ denotes the collection of hyperparameters in the basis correlation functions. A possible choice of prior which expresses weak knowledge about Σ is

$$\pi(\Sigma, \tilde{\delta}) \propto \pi(\tilde{\delta})|\Sigma|^{\frac{k+1}{2}}$$

where $\pi(\tilde{\delta})$ is an arbitrary prior on the correlations length of the basis correlation functions, as discussed in *AltGPPriors*. Note that the hyperparameters $\tilde{\delta}$ do not correspond directly to the correlation lengths of individual outputs, so this prior is not directly equivalent to the weak prior for the separable covariance function given above.

Comparison of the alternatives

The independent outputs approach is perhaps the simplest case, for then the procedure *ProcBuildMultiOutputGP* becomes equivalent to building r separate, independent emulators, one for each output, using the methods set out in the core thread *ThreadCoreGP*. This allows different mean and correlation functions to be fitted for each output, and in particular different outputs can have different correlation length parameters in their hyperparameter vectors. Using the independent emulators for tasks such as prediction, uncertainty and sensitivity analyses is set out in *ThreadGenericMultipleEmulators*, and is also relatively simple.

However, if the number of outputs r is large, building r emulators may be time consuming. Also, the independent outputs approach fails to capture any correlation between outputs, which may be a particular problem if we are interested in a function that combines outputs, as discussed in *ProcPredictMultiOutputFunction*. In that case the easiest non-independent approach is to use a separable covariance function, which leads to several simplifications to the general multivariate emulator, resulting in a simple emulation methodology. This is described in *ProcBuildMultiOutputGPSep*. A drawback of the separable covariance is that it imposes a restriction that all the outputs have the same correlation function $c(.,.)$ across the input space. This means that all outputs have the same correlation length, so the resulting emulator may not perform well if the outputs represent several different types of physical quantity, since it assumes that all outputs have the same smoothness properties.

It is worth noting that outputs which are very highly correlated will necessarily have very similar correlation functions. So in the situation where the independent outputs approach is most inappropriate because we expect the outputs to be strongly correlated, the separable covariance function is more likely to be reasonable.

The situation in which the separable covariance function is most likely to be inappropriate is when multiple outputs represent a variety of types of physical quantities. In that case we may expect there to be some correlation between the outputs due to shared dependence on some underlying processes, but the outputs are likely to have different smoothness properties, so require different correlation functions across the input space. The most flexible approach for such situations is to use a non-independent, nonseparable covariance function.

The downside of nonseparable covariance functions are their complexity: they require a large number of hyperparameters to be estimated before they can be used. Described above are two types of nonseparable covariance function. In practical terms, the main difference between them is the way in which they are parameterised. The convolution covariance function is controlled by hyperparameters $\omega = (\tilde{\Sigma}, \delta)$, where the δ directly control the input space correlation functions but $\tilde{\Sigma}$ gives only limited control over the between-outputs covariance matrix. Conversely, in the LMC covariance function we have parameters $\omega = (\tilde{\delta}, \Sigma)$, where Σ is the between-outputs covariance matrix, but $\tilde{\delta}$ gives only limited control over the input space correlation functions. Therefore, if we have more meaningful prior information about the correlation lengths than the between-outputs covariances, we may favour the convolution covariance function, whereas if we have more meaningful prior information about the between-outputs covariances than the correlation lengths, we may favour the LMC covariance function. In cases where we have weak information about both correlation lengths and the between-outputs covariances, it may be wise to try both and see which performs the best.

We conclude that, when building a multivariate emulator, it would be desirable to try a variety of the above covariance functions, perform diagnostics on the resulting emulators (as described in *ProcValidateCoreGP*), and select that which is most fit for purpose. If the outputs have different correlation lengths, and there is interest in joint predictions of multiple outputs, then the nonseparable covariance functions may be best. However, nonseparable covariance functions may not be practicable in large dimension simulators, so then we must choose between the independent outputs approach and the separable covariance function. The former is likely to be best if interest is confined to marginal output predictions, while the latter may be necessary if joint predictions are required. An alternative approach to treating high dimensional output spaces is to attempt dimension reduction, for example using PCA, and then construct a separable or nonseparable Gaussian process in the reduced dimension space as discussed in *ProcOutputsPrincipalComponents*.

8.130 Alternatives: Prior distributions for multivariate GP hyperparameters

8.130.1 Overview

The prior specification for the *ThreadCoreGP* priors is discussed and alternatives are described in *AltGPPriors*. *ThreadCoreGP* deals with the *core problem* and in particular with emulating a single output - the univariate case. The multivariate case of emulating several outputs is considered in the variant thread *ThreadVariantMultipleOutputs*, and there are a range of possible parameterisations (see the alternatives page on approaches to emulating multiple outputs (*AltMultipleOutputsApproach*)) which require different prior specifications. Some of the alternatives reduce to building independent univariate emulators, for which the discussion in *AltGPPriors* is appropriate. We focus here on the case of a input-output *separable* multi output emulator and we assume that the mean function has the linear form. For further discussion of nonseparable covariances several alternative parameterisations are discussed in *AltMultivariateCovarianceStructures*.

In the case of a input-output *separable* multi output emulator, the prior specification will be over a matrix :math:beta of *hyperparameters<DefHyperparameter>* for the mean function, a between-outputs variance matrix Σ and a vector δ of hyperparameters for the correlation function over the input space.

A fully Bayesian analysis requires hyperparameters to be given prior distributions. We consider here alternative ways to specify prior distributions for the hyperparameters of the multi output problem.

8.130.2 Choosing the Alternatives

The prior distributions should be chosen to represent whatever prior knowledge the analyst has about the hyperparameters. However, the prior distributions will be updated with the information from a set of training runs, and if there is substantial information in the training data about one or more of the hyperparameters then the prior information about those hyperparameters may be essentially irrelevant. In the multi output case it can often make sense to choose prior distributions which are more simple to work with and can accommodate prior beliefs reasonably well.

In general we require a *joint* distribution $\pi(\beta, \Sigma, \delta)$ for all the hyperparameters. Where required, we will denote the marginal distribution of δ by $\pi_\delta(\cdot)$, and similarly for marginal distributions of other groups of hyperparameters.

8.130.3 The Nature of the Alternatives

Priors for Σ

In most applications, there will be information about Σ in the training data, although we note that Σ contains $r(r-1)/2$ unique values in the between outputs variance matrix (where r is the number of outputs), so more training runs might be required to ensure this is well identified from the data compared to the single output case. Unless there is strong prior information available regarding this hyperparameter, it would be acceptable to use the conventional *weak prior* specification

:math:‘pi_{Sigma}(Sigma) \propto | \Sigma |^{-frac{r+1}{2}} ‘

independently of the other hyperparameters.

In situations where the training data are more sparse, which may arise for instance when the simulator is computationally demanding, prior information about Σ may make an important contribution to the analysis.

Genuine prior information about :math:Sigma in the form of a *proper<DefProper>* prior distribution should be specified by a process of *elicitation* - see comments at the end of this page. See also the discussion of conjugate prior distributions below.

Priors for β

As with the univariate case, we would expect to find that in most applications there is enough evidence in the training data to identify β well, particularly when the mean function is specified in the linear form, so that the elements of β are a matrix of regression parameters. Then it is acceptable to use the conventional weak prior specification

:math:‘pi_{beta}(beta) \propto 1 ‘

independently of the other hyperparameters.

If there is a wish to express genuine prior information about β in the form of a proper prior distribution, then this should be specified by a process of elicitation - see comments at the end of this page. See also the discussion of conjugate prior distributions below.

Conjugate priors for β and Σ

When substantive prior information exists and is to be specified for :math:beta and/or Σ, then it is convenient to use *conjugate<DefConjugate>* prior distributions if feasible.

If prior information is to be specified for Σ alone (with the weak prior specification adopted for β), the conjugate prior family is the inverse Wishart family. Elicitation of such distributions is not a trivial matter and will be developed further at a later date.

β is a matrix of regression parameters in a linear form of mean function, and if prior information is to be specified about both β and Σ, then the conjugate prior family is the matrix normal inverse Wishart family. Specifying such a distribution is a complex business and knowledge is still developing in this area.

Although these conjugate prior specifications make subsequent updating using the training data as simple as in the case of weak priors, the details are not given in the *MUCM* toolkit because it is expected that weak priors for β and Σ will generally be used. In the multivariate setting it becomes increasingly difficult to capture beliefs about covariances in a simple manner. Also the number of judgements that have to be made increases quadratically with the dimension of the output space, which makes expert elicitation a real challenge.

If prior information is to be specified for β alone, the conjugate prior family is the matrix normal family, but for full conjugacy the between-columns variance matrix of β should be equal to Σ in the same way as is found in the matrix normal inverse Wishart family. This seems unrealistic when weak prior information is to be specified for Σ, and so we do not discuss this conjugate option further.

Priors for δ

This case is very similar to that discussed in the core *AltGPPriors* thread and we do not repeat it here. We note that in the case of an input - output separable emulator there are no more correlation function parameters to estimate than in the univariate case - the extra complexity is all in β and Σ. For more complex representations, such as the Linear Model of Coregionalisation covariances and convolution covariances, discussed in *AltMultivariateCovarianceStructures*, there are more parameters and prior specification is discussed on that page.

8.130.4 Additional Comments, References, and Links

References to the literature on elicitation of prior distributions may be found at the end of *AltGPPriors*. However, the elicitation of distributions for matrices such as β or Σ is very challenging and there is very little literature to provide guidance. Accordingly, there is relatively little experience of elicitation for multivariate GPs.

8.131 Alternatives: Numerical solution for Karhunen Loeve expansion

8.131.1 Overview

The analytical solutions of the Fredholm equation only exist for some particular processes, that is, for particular covariance functions. For this reason the numerical solutions are usually essential. There are two main methods: the integral method and the expansion method.

8.131.2 Choosing the Alternatives

The eigenfunction is not explicitly defined when using the integral method.Therefore the expansion method is preferred where the eigenfunction is approximated using some orthonormal basis functions.

8.131.3 The Nature of the Alternatives

Integral method

This is a direct approximation for the integral, i.e. $\int_0^1 f(t)dt \approx \sum_{i=0}^{n+1} \omega_i f(t_i)$.

The bigger the n, the better the approximation.

Expansion method

This method is known as Galerkin method. It employs a set of basis functions. The commonly used basis functions are Fourier (trigonometric) or wavelets. Implementation is discussed in *ProcFourierExpansionForKL* and *ProcHaarWaveletExpansionForKL*. Note that although we have presented the methods separately, there are many common features which would comprise a generic method which could be used, in principle, for any approximation based on a particularly basis.

8.131.4 Additional Comments, References, and Links

The solution of Fredholm equations of the second kind has a strong background in applied mathematics and numerical analysis. A useful volume is:

K. E. Atkinson. Numerical solutions of integral equations of the second kind. Cambridge, 1997.

8.132 Alternatives: Optimal design criteria

8.132.1 Overview

Several toolkit operations call for a set of points to be specified in the *simulator* space of inputs, and the choice of such set is called a design. For instance, to create an *emulator* of the simulator we need a set of points, the *training sample*, at which the simulator is to be run to provide data in order to build the emulator.

Optimal design seeks a design which maximises or minimises a suitable criterion relating to the efficiency of the experiment in terms of estimating parameters, prediction and so.

In mainstream optimal design this is based on some function of the variance-covariance matrix $\sigma^2(X^T X)^{-1}$ from a regression model $Y = X\beta + \varepsilon$. It is heavily model based since X depends on the model. Thus, recall that in regression will be built on a vector of functions $f(x)$: so that the mean at a particular x is $\eta(x) = E(Y_x) = \sum \beta_j f_j(x) = \beta^T f(x)$. Then $X = f_j(x_i)$ and $x_1, \cdots, x_n$ is the design.

Below, a term like $\min_{design}$ means minimum over some class of designs, e.g. all designs of sample size n with design points $x_1, \cdots, x_n$ in a design space $\mathcal{X}$. We may standardise the design space to be, for example $[-1, 1]^d$, where d is the number of scalar inputs, but we can have very non standard design spaces.

For details about optimal design algorithms see, *AltOptimalDesignAlgorithms*.

8.132.2 Choosing the Alternatives

Each of the criteria listed below serves a specific use, and some comments are made after each.

8.132.3 The Nature of the Alternatives

Classical Optimality Criteria

D-optimality

D-optimality is used to choose a design that minimises the generalised variance of the estimators of the model parameters namely $\min_{design} det((X^T X)^{-1}) \Leftrightarrow \max_{design} det(X^T X)$. This is perhaps the most famous of optimal design criteria, and has some elegant theory associated with it. Because the determinant is the product of the eigenvalues it is sensitive to X, or equivalently $X^T X$, being close to not being full rank. Conversely it guards well against this possibility: a small eigenvalue implies that the model is close to being "collinear", i.e. X is not of full rank. The criterion has the nice property that a linear reparametrisation (such as shifting and rescaling the parameters) has no effect on the criterion

G-optimality

This criterion used to find the design that minimises (over the design space) the variance of the prediction of the mean $\eta(x)$ at a point x. This variance is equal to $\sigma^2 f(x)^T (X^T X)^{-1} f(x)$ and therefore G-optimality means: minimise the maximum value of this variance: $\min_{design} \max_{x \in \mathcal{X}} f(x)^T (X^T X)^{-1} f(x)$. The General Equivalence Theorem of Kiefer and Wolfowitz says that D- and G-optimality are equivalent (under a general definition of a design as a mass function and over the same design space).

E-optimality

The E-optimality criterion is used when we are interested in estimating any (normalized) linear function of the parameters. It guards against the worst such value: $\min_{design} \max_j \lambda_j((X^T X)^{-1})$. As such it is even stronger than D-optimality in guarding against collinearity. In other words large eigen-values occur when X is close to singularity.

Trace optimality

This is used when the aim of the experiment is to minimize the sum of the variances of the least squares estimators of the parameters: $min_{design} \mathrm{Tr}(X^T X)^{-1}$. It is one of the easiest criteria in conception, but suffers from not being invariant under linear reparametrisations (unlike D-optimality). It is more appropriate when parameter have clear meanings, eg as the effect of a particular factor.

A-optimality:

It is used when the aim of the experiment is to estimate more than one special linear functions of the parameters, e.g. $K\beta$. The least squares estimate has $\mathrm{Var}[K^T \hat{\beta}] = \sigma^2 K^T (X^T X)^{-1} K$. Using the trace criterion, this has trace: $\sigma^2 \mathrm{Tr}(X^T X)^{-1} A$ with $A = KK^T$. Care has to be taken because the criterion is not invariant with respect to reparametrisation, including rescaling, of parameters and because A affects the importance given to different parameters.

c-optimality

This is to minimise the variance of a particular linear function of the parameters: $\phi = \sum c_j \beta_j = c^T \beta$ and $\min_{design} c^T (X^T X)^{-1} c$. It is a special case of A-optimality and has a nice geometric theory of its own. It was one of the earliest studied criteria.

Bayesian optimal experimental design

There is an important general approach to design in the Bayesian setting. The choice of experimental design is a decision which may be made after the specification of any prior distribution on the parameters in a model but before the observations are made (outputs are measured). In this sense it is an pre-posterior decision. Suppose that β refers to the unknown parameters of interest; not just the labeled parameters but may include predicted outputs at special points. Let $\phi(\pi(\beta|Y))$ be a functional on the posterior distribution $\pi(\beta|Y)$ (after the experiment is conducted) which measures some feature of the posterior such as its peakedness. In the case that $\phi = E(L(\hat{\beta}, \beta))$, for some estimator $\hat{\beta}$ and loss function L then ϕ is the posterior Bayes risk of $\hat{\beta}$.

If $\phi = \min_{\hat{\beta}} E(L(\hat{\beta}, \beta))$ then $\hat{\beta}$ is the Bayes estimator with respect to L and we have the (posterior) Bayes risk. This is ideal from the Bayes standpoint although it may be computationally easier to use a simpler non-Bayes estimator but still compute the Bayes risk.

The full Bayesian optimal design criterion with respect to ϕ is $\min_{design} E_Y \phi(\pi(\beta|Y))$, where Y is the output generated by the experiment. Here, of course, the distribution of Y is affected by the design. In areas such as non-linear regression one make be able to compute a local optimal design using a classical estimator such as the maximum likelihood estimator. In such a case the ϕ value may depend on this unknown β: $\phi(\beta)$. An approximate Bayesian criteria is then $\min_{design} E_\beta(\phi(\beta))$, where the expectation is with respect to the prior distribution on β, $\pi(\beta)$. The approximate full Bayes criteria (which is computationally harder) and approximate Bayes criteria can give similar solutions.

There are Bayesian analogues of all the classical optimal design criteria listed above. The idea is to replace the variance matrix of the least squares estimates of the regression parameter β, by the posterior variance matrix $\text{Var}[\beta|Y]$. Thus, if we take the standard regression model: $Y = X\beta + \varepsilon$ and let $\beta \sim N(\mu, \sigma^2 I)$ and $\mu \sim N(0, \Gamma)$, then $\text{Var}[\beta|Y] = (\sigma^{-2} X^T X + \Gamma^{-1})^{-1}$.

Important note: in this simple case the posterior covariance does not depend on the observed data, so that the prior expectation E_Y in the Bayes rule for design, is not needed.

Bayesian Information criteria

The information-theoretical approach to experimental design goes some way towards being an all purpose philosophy. It is easiest to explain by $\phi(\pi)$ in the Bayes formulation to be Shannon entropy. For a general random variable X with pdf $p(x)$ this is $\text{Ent}(X) = -E_X(\log(p(X)) = -\int \log(p(x))p(x)dx$.

Shannon information is $Inf(X) = -\text{Ent}(X)$.

The information approach is to minimise the expected posterior entropy.

$$\min_{design} E_Y \text{Ent}(\beta|Y)$$

This is often expressed as: maximise the expected information gain $E_Y(G)$, where: $G = Inf(\beta|Y) - Inf(\beta)$, where the second term on the right hand side is the prior information. An important result says that $E_Y(G)$ is always non-negative, although in actual experiment cases G may decrease. This result can be generalised to a wider class of information criteria which include Tsallis entropy $E_X\left(\frac{p(X)^\alpha - 1}{\alpha}\right), \quad \alpha > -1$.

Maximum Entropy Sampling (MES)

This is a special way of using the Bayesian information criterion for prediction. We exploit an importat formula for Shannon entropy which applies to two random variables: (U, V):

$$\text{Ent}(U, V) = \text{Ent}(U) + \text{E}_U(\text{Ent}(V|U).$$

Let S represent a set of candidate points, which covers the whole design space well. This could for example be a large factorial design, or a large space-filling design. Let and let s be a possible subset of s, to be use as a design. Then the complement of s namely s can be thought of as the "unsampled" points.

Then partition Y: $(Y_s, Y_{S\setminus s})$. Then for prediction we could consider: $\text{Ent}(Y_s, Y_{S\setminus s}) = \text{Ent}(Y_s) + \text{E}_{Y_s}[\text{Ent}(Y_{S\setminus s}|Y_s)]$. The joint entropy on the left hand side is fixed, that is does not depend on the choics of design. Therefore, since the Bayes criterion is to minimise, over the choice of design, the second term on the right, it is optimal to maximise the first term on the right. This is simply the entropy of the sample, and typically requires no conditioning computation. For the simple Bayesian Gaussian case we have

$$\max_{design} |R + X\Gamma X^T|.$$

Where R is the variance matrix of the process part of the model, X is the design matrix for the regression part of the model and Γ is prior variance of the regression parameters (as above).

We see that $Y_{S\setminus s}$ plays the role of the unknown parameter in the general Bayes formulation. This is familiar in the Bayesian context under the heading *predictive distributions*. To summarise: select the design to achieve mimumum entropy (= maximum information) of the joint predictive distribution for all the unsampled points, by maximising the entropy of the sampled points.

Integrated Mean Squared Error (IMSE)

This criterion aims at minimising the mean squared error of prediction over the unsampled points. As for MES, above, this is based on the predictive distribution for the unsampled points.

The mean squared prediction error (MSPE), under the standard model at a single point is given by

$$\text{MSPE}(\hat{Y}(x)) = \sigma^2 \left[1 - \begin{pmatrix} f(x)^T & r(x)^T \end{pmatrix} \begin{bmatrix} 0 & F^T \\ F & R \end{bmatrix} \begin{pmatrix} f(x) \\ r(x) \end{pmatrix} \right]$$

Several criteria could be based on this quantity as the point x ranges over the design space $\mathcal{X}$.

The integrated mean squared prediction error, which is the predictive error averaged over, the design space, $\mathcal{X}$,

$$J(\mathcal{D}) = \int_{\mathcal{X}} \frac{\text{MSPE}[\hat{Y}(x)]}{\sigma^2} \omega(x) dx$$

where $\omega(\cdot)$ is a specified non-negative weight function satisfying $\int_{\mathcal{X}} \omega(x)dx$.

This criterion has been favoured by several authors in computer experiments, (Sacks et al,1989).

Other criteria are (i) minimising over the design the maximum MSPE over the unsampled points (ii) minimising the biggest eigenvalues of the predictive (posterior) covariance matrix. Note that each of the above criteria is the analogue of a classical criterion in which the parameters are replaced by the values of the process at the unsampled points in the candidate set: thus Maximum Entropy Sampling (in the Gaussian case) is the analogue of D-optimality, IMSE is a type of trace-optimality and (i) and (ii) above are types of types of G- and E-optimality respectively.

Bayesian sequential optimum design

The Bayesian paradigm is very useful in understanding sequential design.

After we have selected a criterion ϕ, see above, the first stage design is to $\min_{design_1} E_{Y_1} \phi(\pi(\beta|Y_1))$, where Y_1 is the output generated in the first stage experiment.

The (naive/myopic) one-step-ahead method is to take the prior distribution (process) at stage 2 to be the posterior process having observed Y_1 and procead to choose the stage 2 design to minimise $\min_{design_2} E_{Y_2} \phi(\pi(\beta|Y_1, Y_2))$,, and so on through further stages, if necessary.

The full sequential rule, which is very difficult to implement, uses the knowledge that one will use an optimal design at future stages to adjust the “risk” at the first stage. For two stages this would give, working backwards: $R_2 = \min_{design_2} E_{Y_2} \phi(\pi(\beta|Y_1, Y_2))$, and then at the first stage choose the first design to achieve $\min_{design_1} E_{Y_1}(R_2)$. The general case is essentially dynamic programming (Bellman rule) and with a finite horizon N the scheme would look like:

$$\min_{design_1} E_{Y_1} \min_{design_2} E_{Y_2} \ldots \min_{design_{N-1}} E_{Y_{N-1}} \phi(\pi(\beta|Y_1,\ldots,Y_N))$$

This full version of sequential design is very "expensive" because the internal expectations require much numerical integration.

One can perform global optimisation to obtain an optimal design over a design space $\mathcal{X}$, but it is convenient to use a large candidate set. As mentioned, this candidate set is typically taken to be a *full factorial design* or a large *space-filling design*. In block sequential design one may add a block optimally which may itself be a smaller space-filling design.

It is useful to stress again notation for the design used in the MES method, above. Thus let S be the candidate set and s the design points and $\bar{s} = S \setminus s$ the unsampled points. Then the optimal design problem is an optimal subset problem: choose $s \subset S$ in an optimal way. This notation helps to describe the important class of *exchange algorithms* in where points are exchanged between s and $\bar{s}$.

A one-point-at-a-time myopic sequential design based on the MES principle places each new design point at the point with maximum posterior variance from the design to date. See also some discussion in *DiscCoreDesign*.

8.132.4 Additional Comments, References, and Links

There is a large literature on algorithms for optimal design and algorithms are incorporated into commercial software, with the most prevalent being algorithms for *D*-optimality. See, also *AltOptimalDesignAlgorithms*.

A. C. Atkinson and A. N. Donev. Optimum Experimental Designs. Oxford Statistical Science Series, vol. 8. Oxford: Clarendon Press, 1992.

A. C. Atkinson, A. N. Donev and R. D. Tobias. Optimum Experimental Designs, with SAS. 2007 Oxford, Oxford University Press. 207.

F. Pukelsheim. Optimal Design of Experiments Friedrich Pukelsheim,Siam: Classics in Applied Mathematics 50. 2006 Original publication: Wiley, 1993.

J.Sacks, W. J. Welch, T. J. Mitchell, & H. P. Wynn, Design and analysis of computer experiments, Statistical Science, 4(4):409-423, 1989.

JMP (SAS product): http://www.jmp.com/software/

K. Chaloner, and I. Verdinelli Bayesian experimental design: a review. Statistical Science, 10, 3, 273–304, 1995.

M. C Shewry and H. P. Wynn. Maximum entropy sampling. Journal of Applied Statistics, 14: 165-170, 1987.

MATLAB: Design of Experiments (Statistics Toolbox$^{\text{TM}}$)

8.133 Alternatives: Optimal Design Algorithms

8.133.1 Overview

Finding an optimal design is a special type of problem, but shares features with more general optimisation problems.Typically the objective function will have very many local optimal so that any of the general methods of global optimization may be used such simulated annealing (SA), genetic algorithims (GA) and global random search. Each of these has a large litertature and many implementations. As with general optimisation a reasonable strategy is to use an algorithm which is good for unimodal problems, such as sequential quadratic programming (SQP), but with multi-start. Another strategy is to start with a global method and finish with a local method. Although stochastic algorithms such as GA have theory which show that under certain conditions the algorithm will converge in a probability sense to a global optimum, it is safer to treat them as simply finding local optima.

8.133.2 Choosing the Alternatives

For small problems, that is, low dimension and low sample size the Branch and Bound algorithm is guaranteed to converge to a global optimum. The technical issue then is to find the "bounds" and "branching" required in the specification of the algorithm. For Maximum Entropy Sampling (MES) spectral bounds are available.

As pointed out, an optimum design problem can be thought of as an optimal subset problem, so that exchange algorithms in which "bad" points are exchanged for "good" are a natural are a natural method and turn out to be fast and effective. But, as with general global optimisation methods, they are not guaranteed to converge to an optimum so multi-start is essential. Various versions are possible such as exchanging more than one point and incorporating random search. In the latter case exchange algorithms share have some of the flavour of global optimisation algorithms which also have randomisation.

The notion of "greadiness" is useful in optimisation algorithms. A typical type of greedy algorithms will move one factor at a time, to find an optimum along that direction, cycling through the factors one at a time. Only under very special consitions are such algorithms guaranteed to converge, but combined with multi-start they can be effective and they are easy to implement.On can think of an exchange algorithms as a type of greedy algorithm.

When the problem has been made discrete by optimally choosing a design from and candidate set it is appropriate to call an algorithm a "combinatorial algorithm": one is choosing a combination of levels of the factors.

8.133.3 The Nature of the Alternatives

We describe two algorithms

Exchange algorithms

This is described in more detail in *ProcExchangeAlgorithm*. The method is to find an optimal design with fixed sample size n. Given a large candidate set and an initial design, which is a subset of the candidate set, points are exchanged between the design and the points in the candidate set, but outside the design. This is done in a way to keep the same size fixed between exchanges. During this exchange one tries to improve the value of the design optimality criterion. Variations are to exchange more than one point, that is a "block" of points. Since the candidate set is large one may select points to be exchanged at random.

Branch and bound algorithms

This is descibed in more detail in *ProcBranchAndBoundAlgorithm*. Branch and Bound searches on a tree of possible designs. The branching tells us how to go up and down the tree. The bounding is a way to avoid having to search a whole branch of the tree by using a bound to show that all designs down a branch are worse than the current branch. In this way the decision at each stage is to carry on down a "live" branch, or jump to a new branch because the rest of a branch is no longer live. The bounds also supply the stopping rule. Roughly, if the algorithm stops because the bound tells us not to continue down a branch and there are no other live branches available then it is straightfoward to see we have a global optimum. The algorithm works well when the tree is set up not to be too enormous and the bound is easy to compute: fast bounding, not too many branches. But there is a tension: the tighter the bound the more effective it is in eliminating branches but typically the harder to find or compute. The hope is that there is a very natural bound and this is often the case with design problems since there is usual matrix theory involved one can use matrix inequalities of various kinds. The optimum subset nature of the problem also lends itself to trees based on unions and intersections of subsets.

8.133.4 Additional Comments, References, and Links

Global optimisation can be used as a generic term, which covers all methods not tailored to particular classes of function, eg convex.

A. Zhigljavsi and A. Zilinskas. Stochastic Global Optimization. Springer, (2008).

8.134 Alternatives: Deciding which screening method to use

We first provide a classification of *screening* methods, followed by a discussion under what settings each screening method may be appropriate.

8.134.1 Classification of screening methods

Screening methods have been broadly categorised in the following categories:

1. **Screening Design** methods. An experimental *design* is constructed with the express aim of identifying *active* factors. This approach is the classical statistical method, and is typically associated with the Morris method (see the procedure page on the Morris screening method (*ProcMorris*) which defines a design optimised to infer the global importance of *simulator* inputs to the simulator output)
2. **Ranking** methods. Input variables are ranked according to some measure of association between the simulator inputs and outputs. Typical measures considered are correlation, or partial correlation coefficients between simulator inputs and the simulator output. Other non-linear measures of association are possible, but these methods tend not to be widely used.
3. **Wrapper** methods. An emulator is used to assess the predictive power of subsets of variables. Wrapper methods can use a variety of search strategies:
    1. Forward selection where variables are progressively incorporated in larger and larger subsets.
    2. Backward elimination where variables are sequentially deleted from the set of active inputs, according to some scoring method, where the score is typically the root mean square prediction error of the simulator output (or some modification of this such as the Bayesian information criterion).
    3. Efroymsons algorithm also known as stepwise selection, proceeds as forward selection but after each variable is added, the algorithm checks if any of the selected variables can be deleted without significantly affecting the Residual Sum of Squares (RSS).
    4. Exhaustive search where all possible subsets are considered.
    5. Branch and Bound strategies eliminates subset choices as early as possible by assuming the performance criterion is monotonic, i.e. the score improves as more variables are added. For a discussion of the algorithm see its procedure page *ProcBranchAndBoundAlgorithm*.
4. **Embedded** methods. For both variable ranking and wrapper methods, the emulator is considered a perfect black box. In embedded methods, the variable selection is integrated as part of the training of the emulator, although this might proceed in a sequential manner, to allow some benefits of the reduction in input variables considered.

In this thread we focus on methods most appropriate for computer experiments that are the most general, i.e. the assumptions made are not overly restrictive to a particular class of models.

8.134.2 Decision process

If the simulator is available and deterministic a screening design approach, the Morris method (see *ProcMorris*), is most appropriate where a one factor at a time (OAT) design is used to identify active inputs. The Morris method is a very simple process, which can be understood as the construction of a design to estimate the expected value and variance (over the input space) of the partial derivatives of the simulator output with respect the simulator inputs. The method creates efficient designs to estimate these, thus to use the method it will be necessary to evaluate the simulator over the Morris design and the method cannot be reliably applied to data from other designs. The restriction to deterministic simulators (where the output is identical for repeated evaluations at the same inputs) is because partial derivatives are likely to be rather sensitive to noise on the simulator output, although this will depend to some degree on the signal to noise ratio in the outputs.

If the simulator is not easily evaluated (maybe simply because we don't have direct access to the code), or the training design has already been created, then design based approaches to emulation are not possible and the alternative methods described above must be considered. Also if the simulator output has a random component for a fixed input, then the below methods can be readily applied.

If the critical features of the simulator output can be captured by a small set of fixed basis functions (often simply linear or low order polynomials) then a regression (wrapper) analysis can be used to identify the active inputs. One such procedure is described in the procedure page for the empirical construction of a BL emulator (*ProcBuildCoreBLEmpirical*) and is an example of a wrapper method. Many other methods for variable selection are possible and can be found in textbooks on statistics and in papers on stepwise variable selection for regression models.

An alternative to the wrapper methods above is to employ an embedded method, Automatic Relevance Determination (ARD), which is described in the procedure page (*ProcAutomaticRelevanceDetermination*). Automatic relevance determination essentially uses the estimates of the input variable length scale hyperparameters in the emulator covariance function to assess the relevance of each input to the overall emulator model. The method has the advantage that the relatively flexible Gaussian process model is employed to estimate the impact of each input, as opposed to a linear in parameters regression model, but the cost is increased computational complexity.

8.134.3 References

Guyon, I. and A. Elisseeff (2003). *An introduction to variable and feature selection*, http://jmlr.csail.mit.edu/papers/volume3/guyon03a/guyon03a.pdf. Journal of Machine Learning Research 3, 1157 - 1182.

8.135 Definition of Term: Active input

A *simulator* will typically have many inputs. In general, the larger the number of inputs the more complex it becomes to build a good *emulator*, and the larger the number of simulator runs required for the *training sample*. There is therefore motivation for reducing the number of inputs.

Fortunately, it is also generally found that not all inputs affect the output(s) of interest appreciably over the required range of input variation. The number of inputs having appreciable impact on outputs for the purpose of the analysis is often small. The term 'active input' loosely means one which does have such an impact. More precisely, the active inputs are those that are deemed to be inputs when building the emulator.

8.136 Definition of Term: Adjoint

An adjoint is an extension to a *simulator* which produces derivatives of the simulator output with respect to its inputs.

Technically, an adjoint is the transpose of a tangent linear approximation; written either by hand or with the application of automatic differentiation, it produces partial derivatives in addition to the standard output of the simulator. An adjoint is computationally more expensive to run than the standard simulator, but efficiency is achieved in comparison to the finite differences method to generating derivatives. This is because where computing partial derivatives by finite differences requires at least two runs of the simulator for each input parameter, the corresponding derivatives can all be generated by one single run of an appropriate adjoint.

8.137 Definition of Term: Assessment

When dealing with an uncertain quantity, the term “assessment” is used to refer to the process of identifying a suitable probabilistic form to represent the current knowledge/uncertainty related to that quantity. This process may for example involve *elicitation*, or other more formal methods.

The term is often used in the context of the assessment of the *model discrepancy*.

8.138 Definition of Term: Bayes linear adjustment

In the context of the *Bayes linear* approach, suppose that we have specified prior means, variances and covariances for a collection of uncertain quantities. If the values of some of the quantities become known, then we reassess our judgements about the remaining quantities. As the Bayes linear approach is *Bayesian*, we modify our prior beliefs in light of the data, which (under the Bayes linear approach) yields the *adjusted expectation* and *adjusted variance* for each quantity and the *adjusted covariance* for every pair.

See the procedure page on calculation of adjusted expectation and variance (*ProcBLAdjust*) for details on calculating the adjusted expectations, variances and covariances, and the discussion page on theoretical aspects of Bayes linear (*DiscBayesLinearTheory*) for further discussion of the theory behind Bayes linear.

8.139 Definition of Term: Bayes linear variance learning

Learning about the variance σ^2 of a complex model can be a challenging problem. The Bayes Linear approach is to make a standard Bayes Linear *adjustment* of our prior beliefs about σ^2 using the observed sample variance and exploiting certain weak assumptions about model behaviour (*second-order exchangeability*).

8.140 Definition of Term: Basis functions

The mean function of an emulator, $m(x)$, expresses the global behaviour of the computer model and can be written in a variety of forms as described in the alternatives page on emulator prior mean function (*AltMeanFunction*). When the mean function takes a linear form it can be expressed as $m(x) = \beta^T h(x)$ where $h(x)$ is a q-vector of known *basis functions* of x which describe global features of the simulator's behaviour and β is a vector of unknown coefficients.

The task of specifying appropriate forms for $h(x)$ is addressed in the alternatives page on basis functions for the emulator mean (*AltBasisFunctions*).

8.141 Definition of Term: Bayes linear

Like the fully *Bayesian* approach in *MUCM*, the Bayes linear approach is founded on a personal (or subjective) interpretation of probability. Unlike the Bayesian approach, however, probability is not the fundamental primitive concept in the Bayes linear philosophy. Instead, expectations are primitive and updating is carried out by orthogonal projection. In the simple case of a vector of real quantities, some of which are observed, this projection is equivalent to minimising expected quadratic loss over a linear combination of the observed quantities, and the adjusted mean and variance that result are computationally the same as the Bayesian conditional mean and variance of a Gaussian vector. If the observation vector consists of the indicator functions for a partition, then the Bayes linear adjustment is equivalent to Bayesian conditioning on the partition.

Further details on the theory behind the Bayes linear approach are given in the discussion page on theoretical aspects of Bayes linear (*DiscBayesLinearTheory*).

8.142 Definition of Term: Bayesian

The adjective ‘Bayesian’ refers to ideas and methods arising in the field of Bayesian Statistics.

Bayesian statistics is an approach to constructing statistical inferences that is fundamentally and philosophically different from the approach that is more commonly taught, known as frequentist statistics.

In Bayesian statistics, all inferences are probability statements about the true, but unknown, values of the parameters of interest. The formal interpretation of those probability statements is as the personal beliefs of the person providing them.

The fact that Bayesian inferences are essentially personal judgements has been the basis of heated debate between proponents of the Bayesian approach and those who espouse the frequentist approach. Frequentists claim that Bayesian methods are subjective and that subjectivity should have no role in science. Bayesians counter that frequentist inferences are not truly objective, and that the practical methods of Bayesian statistics are designed to minimise the undesirable aspects of subjectivity (such as prejudice).

As in any scientific debate, the arguments and counter-arguments are many and complex, and it is certainly not the intention of this brief definition to go into any of that detail. In the context of the MUCM toolkit, the essence of an *emulator* is that it makes probability statements about the outputs of a *simulator*, and the frequentist approach does not formally allow such statements. Therefore emulators are necessarily Bayesian.

In the *MUCM* field, there is a recognised alternative to the fully Bayesian approach of characterising uncertainty about unknown parameters by complete probability distributions. This is the *Bayes linear* approach, in which uncertainty is represented only through means, variances and covariances.

8.143 Definition of Term: Best Input

No matter how careful a particular model of a real system has been formulated, there will always be a discrepancy between reality, represented by the system value y, and the *simulator* output $f(x)$ for any valid input x. Perhaps, the simplest way of incorporating this *model discrepancy* into our analysis is to postulate a best input x^+, representing the best fitting values of the input parameters in some sense, such that the difference $d = y - f(x^+)$ is independent or uncorrelated with f, $f(x^+)$ and x^+.

While there are subtleties in the precise interpretation of x^+, especially when certain inputs do not correspond to clearly defined features of the system, the notion of a best input is vital for procedures such as *calibration*, history matching and prediction for the real system.

8.144 Definition of Term: Calibration

The inputs to a *simulator* that are the correct or best values to use to predict a particular real-world system are very often uncertain.

Example: An atmospheric dispersion simulator models the way that pollutants are spread through the atmosphere when released. Its outputs concern how much of a pollutant reaches various places or susceptible targets. In order to use the simluator to predict the effects of a specific pollutant release, we need to specif the appropriate input values for the location of the release, the quantity released, the wind speed and direction and so on. All of these may in practice be uncertain.

If we can make observations of the real-world system, then we can use these to learn about those uncertain inputs. A crude way to do this is to adjust the inputs so as to make the simulator predict as closely as possible the actual observation points. This is widely done by model users, and is called calibration. The best fitting values of the uncertain parameters are then used to make predictions of the system.

Example: Observing the amount of pollutant reaching some specific points, we can then calibrate the model and use the best fitting input values to predict the amounts reaching other points and hence assess the consequences for key susceptible targets.

In *MUCM*, we take the broader view that such observations allow us to learn about the uncertain inputs, but not to eliminate uncertainty. We therefore consider calibration to be the process of using observations of the real system to modify (and usually to reduce) the uncertainty about specific inputs.

See also *data assimilation*.

8.145 Definition of Term: Code uncertainty

An *emulator* is a probabilistic representation of the output(s) of a *simulator*. For any given input configuration the output is uncertain (unless this input configuration was included in the *training sample*). This uncertainty is known as code uncertainty. The emulator defines exactly what the code uncertainty is about the output(s) from any given configuration(s) of inputs.

Because the output(s) are uncertain, we are also uncertain about properties of those outputs, such as the uncertainty mean in an *uncertainty analysis*. The code uncertainty expressed in the emulator then implies code uncertainty about the properties of interest. Statements that we can make about those properties are therefore *inferences*. For instance, we may estimate a property by its mean, which is evaluated with respect to its code uncertainty distribution.

8.146 Definition of Term: Conjugate prior

In the *Bayesian* approach to statistics, a prior distribution expresses prior knowledge about parameters in a statistical analysis. The prior distribution is then combined with the information in the data using Bayes' theorem, and the resulting posterior distribution is used for making inference statements (such as estimates or credible intervals) about the parameters. Specification of the prior distribution is therefore an important task in Bayesian statistics. Two simplifying techniques that are often used are to employ *weak* prior distributions (that represent prior information that is supposed to be weak relative to the information in the data) and conjugate prior distributions.

A conjugate distribution is of a mathematical form that combines conveniently with the information in the data, so that the posterior distribution is easy to work with. The specification of prior information is generally an imprecise process, and the particular choice of distributional form is to some extent arbitrary. (It is this arbitrariness that is objected to by those who advocate the *Bayes linear* approach rather than the fully Bayesian approach; both may be found in the toolkit.) So as long as the conjugate form would not obviously be an inappropriate representation of prior information it is sensible to use a conjugate prior distribution.

8.147 Definition of Term: Correlation length

The degree of *smoothness* of an *emulator* is determined by its correlation function. Smoothness is in practice controlled by *hyperparameters* in the correlation function that define how slowly the correlation between the simulator outputs at two input points declines as the distance between the points increases. These hyperparameters are typically called correlation length parameters. Their precise role in determining smoothness depends on the form of correlation function.

8.148 Definition of Term: Data Assimilation

Data assimilation is a term that is widely used in the context of using observations of the real-world process to update a *simulator*. It generally applies to a *dynamic* simulator that models a real-world process that is evolving in time. At each time step, the simulator simulates the current state of the system as expressed in the *state vector*. In data assimilation, observation of the real-world process at a given time point is used to learn about the true status of the process and hence to adjust the state vector of the simulator. Then the simulator's next time step starts from the adjusted state vector, and should therefore predict the system state better at subsequent time steps.

Data assimilation is thus also a dynamic process. It is similar to *calibration* in the sense that it uses real-world observations to learn about simulator parameters, but the term calibration is generally applied to imply a one-off learning about fixed parameters/inputs.

8.149 Definition of Term: Decision-based sensitivity analysis

In decision-based *sensitivity analysis* we consider the effect on a decision which will be based on the output $f(X)$ of a *simulator* as we vary the inputs X, when the variation of those inputs is described by a (joint) probability distribution. This probability distribution can be interpreted as describing uncertainty about the best or true values for the inputs.

We measure the sensitivity to an individual input X_i by the extent to which we would be able to make a better decision if we could remove the uncertainty in that input.

A decision problem is characterised by a set of possible decisions and a utility function that gives a value $U(d, f(x))$ if we take decision d and the true value for the input vector is x. The optimal decision, given the uncertainty in X, is the one which maximises the expected utility. Let U^* be the resulting maximised expected utility based on the current uncertainty in the inputs.

If we were to remove the uncertainty in the i-th input by learning that its true value is $X_i = x_i$, then we might make a different decision. We would now take the expected utility with respect to the *conditional* distribution of X given that $X_i = x_i$, and then maximise this with respect to the decision d. Let $U_i^*(x_i)$ be the resulting maximised expected utility. This of course depends on the true value x_i of X_i, which we do not know. The decision-based sensitivity measure for the i-th input is then the value of learning the true value of X_i in terms of improved expected utility, i.e. $V_i = \mathrm{E}[U_i^*(X_i)] - U^*$, where the expectation in the first term is with respect to the marginal distribution of X_i.

We can similarly define the sensitivity measure for two or more inputs as being the value of learning the true values of all of these.

Variance-based sensitivity analysis is a special case of decision-based analysis, when the decision is simply to estimate the true output $f(X)$ and the utility function is negative squared error. In practice, though, variance-based sensitivity analysis provides natural measures of sensitivity when there is no specific decision problem.

8.150 Definition of Term: Design

In order to build an *emulator* for a *simulator*, we need data comprising a *training sample* of simulator runs. The training sample design is the set of points in the space of simulator inputs at which the simulator is run.

There are a number of related design contexts which arise in the *MUCM* toolkit.

- A validation sample design is a set of points in the input space at which additional simulator runs are made to *validate* the emulator.
- In the context of *calibrating* a simulator, we may design an observational study by specifying a set of points in the space of input values which we can control when making observations of the real-world process.

- Where we have more than one simulator available, a design may specify sets of points in the input spaces of the various simulators.
- For complex tasks using the emulator, a general solution is via simulated realisations of the output function $f(\cdot)$. For this purpose we need a set of input configurations at which to simulate outputs, which we refer to as a realisation design.

Some general remarks on design options for the core problem are discussed in the alternatives page on training sample design for the core problem (*AltCoreDesign*).

8.151 Definition of Term: Deterministic

A *simulator* is referred to as deterministic if, whenever it is run with a given input configuration it gives the same output(s).

8.152 Definition of Term: Dynamic simulator

A dynamic *simulator* models a real-world process that evolves over time (or sometimes in space, or both time and space). Its inputs typically include an initial *state vector* that defines the initial state of the process, as well as other fixed inputs governing the operation of the system. It often also has external *forcing inputs* that provide time-varying impacts on the system. The dynamic nature of the simulator means that as it runs it updates the state vector on a regular time step, to model the evolving state of the process.

Example: A simulator of a growing plant will have a state vector describing the sizes of various parts of the plant, the presence of nutrients and so on. Fixed inputs parameterise the biological processes within the plant, while forcing inputs may include the temperature, humidity and intensity of sunlight at a given time point.

At time t, the simulator uses the time t value of its state vector, the fixed inputs and the time t values of the forcing inputs to compute the value of its state vector at time $t + \Delta t$ (where Δt is the length of the simulator's time step). It then repeats this process using the new state vector, the fixed inputs and new forcing input values to move forward to time $t + 2\Delta t$, and so on.

8.153 Definition of Term: Elicitation

Probability distributions are needed to express uncertainty about unknown quantities in various contexts. In the *MUCM* toolkit, there are two principal areas where such probability distributions may be required. One is to specify prior knowledge about *hyperparameters* associated with building a *Gaussian process emulator*, while the other is to formulate uncertainty about inputs to the *simulator* (for instance, for the purposes of *uncertainty* or *sensitivity* analyses). These probability distributions may be created in various ways, but the most basic is to represent the knowledge/beliefs of a relevant expert.

The process of formulating an expert's knowledge about an uncertain quantity as a probability distribution is called elicitation.

8.154 Definition of Term: Emulator

An emulator is a statistical representation of a *simulator*. For any given configuration of input values for the simulator, the emulator provides a probabilistic prediction of one or more of the outputs that the simulator would produce if it were run at those inputs.

Furthermore, for any set of input configurations, the emulator will provide a joint probabilistic prediction of the corresponding set of simulator outputs.

Example: A simulator of a nuclear power station reactor requires inputs that specify the flow of gas through the reactor, and produces an output which is the steady state mean temperature of the reactor core. A simulator of this output would provide probabilistic predictions of the mean temperature (as output by the actual simulator) at any single configuration of gas flow inputs, or at any set of such configurations.

The probabilistic predictions may take one of two forms depending on the approach used to build the emulator. In the fully *Bayesian* approach, the predictions are complete probability distributions.

Example: In the previous example, the fully Bayesian approach would provide a complete probability distribution for the output from any single configuration of inputs. This would give the probability that the output (mean temperature) lies in any required range. In particular, it would also provide any desired summaries of the probability distribution, such as the mean or variance. For a given set of input configurations, it would produce a joint probability distribution for the corresponding set of outputs, and in particular would give means, variances and covariances.

In the *Bayes linear* approach, the emulator's probabilistic specification of outputs comprises (adjusted) means, variances and covariances.

Example: The Bayes linear emulator in the above example would provide the (adjusted) mean and (adjusted) variance for the simulator output (mean temperature) from a given single configuration of gas flow inputs. When the emulator is used to predict a set of outputs from more than one input configuration, it will also provide (adjusted) covariances between each pair of outputs.

Strictly, these 'adjusted' means, variances and covariances have a somewhat different meaning from the means, variances and covariances in a fully Bayesian emulator. Nevertheless, they are in practice interpreted the same way - thus, the (adjusted) mean is a point estimate of the simulator output and the square-root of the (adjusted) variance is a measure of accuracy for that estimate in the sense of being a root-mean-square distance from the estimate to the true simulator output. In practice, we would drop the word 'adjusted' and simply call them means, variances and covariances, but the distinction can be important.

8.155 Definition of Term: Exchangeability

Exchangeability is regarded as a fundamental concept in any statistical modelling – regardless of whether that analysis is performed from a *Bayesian* or frequentist perspective. A sequence of random variables is described as exchangeable if our beliefs about that collection are unaffected by the order in which they appear. For example, if we perform 100 tosses of a coin, then, if the coin tosses are exchangeable, our beliefs about the fairness of the coin will be unaffected by the order in which we observe the heads and tails.

In formal terms, when our beliefs about a sequence of random variables $X_1, X_2, \ldots$ are characterised by a joint probability distribution, if the X_is are exchangeable then the joint probability distribution of the sequence is the same as the joint distribution over any re-ordering of the X_is. In fact, the assertion of exchangeability implies the existence of this underlying joint distribution.

This idea of exchangeability underpins the concept of prediction, as when past observations and future observations are exchangeable then the future will be predictable on the basis of the data gathered in the past.

8.156 Definition of Term: Forcing Input

A *dynamic simulator* models a process that is evolving, usually in time. Many dynamic simulators have forcing inputs that specify external influences on the process that vary in time. In order to simulate the evolution of the process at each point in time, the simulator must take account of the values of the forcing inputs at that time point.

Example: A simulator of water flow through a river catchment will have forcing inputs that specify the rainfall over time. Humidity may be another forcing input, since it will govern the rate at which water in the catchment evaporates.

8.157 Definition of Term: Gaussian process

A Gaussian process (GP) is a probability model for an unknown function.

If a function f has argument x, then the value of the function at that argument is $f(x)$. A probability model for such a function must provide a probability distribution for $f(x)$, for any possible argument value x. Furthermore, if we consider a set of possible values for the argument, which we can denote by $x_1, x_2, ..., x_N$, then the probability model must provide a joint probability distribution for the corresponding function values $f(x_1), f(x_2), \cdots, f(x_N)$.

In *MUCM* methods, an *emulator* is (at least in the fully Bayesian approach) a probability model for the corresponding *simulator*. The simulator is regarded as a function, with the simulator inputs comprising the function's argument and the simulator output(s) comprising the function value.

A GP is a particular probability model in which the distribution for a single function value is a normal distribution (also often called a Gaussian distribution), and the joint distribution of a set of function values is multivariate normal. In the same way that normal distributions play a central role in statistical practice by virtue of their mathematical simplicity, the GP plays a central role in (fully Bayesian) MUCM methods.

Just as a normal distribution is identified by its mean and its variance, a GP is identified by its mean function and its covariance function. The mean function is E$[f(x)]$, regarded as a function of x, and the covariance function is Cov$[f(x_1), f(x_2)]$, regarded as a function of both x_1 and x_2. Note in particular that the variance of $f(x)$ is the value of the covariance function when both x_1 and x_2 are equal to x .

A GP emulator represents beliefs about the corresponding simulator as a GP, although in practice this is always conditional in the sense that the mean and covariance functions are specified in terms of uncertain parameters. The representation of beliefs about simulator output values as a GP implies that the probability distributions for those outputs are normal, and this is seen as an assumption or an approximation. In the Bayes linear approach in MUCM, the assumption of normality is not made, and the mean and covariance functions alone comprise the Bayes linear emulator.

8.158 Definition of Term: History Matching

History matching is a process that involves identifying the set of all inputs x that would give rise to acceptable matches between the simulator output $f(x)$ and the observed data z.

Often the process involves the iterative removal of regions of the input space by the application of cutoffs to various *implausibility measures*.

8.159 Definition of Term: Hyperparameter

A parameter is any variable in a probability distribution. For instance, a normal distribution is generally defined to have two parameters, its mean and its variance.

The term hyperparameter is used (fairly loosely) in Bayesian statistics to mean a parameter in the prior distribution.

For the *Gaussian process emulator* the term denotes the parameters in the *mean* or *variance* functions.

8.160 Definition of Term: Implausibility Measure

An implausibility measure is a function $I(x)$ defined over the whole input space which, if large for a particular x, suggests that there would be a substantial disparity between the simulator output $f(x)$ and the observed data z, were we to evaluate the model at x.

In the simplest case where $f(x)$ represents a single output and z a single observation, the univariate implausibility would look like:

$$I^2(x) = \frac{(\mathrm{E}[f(x)] - z)^2}{\mathrm{Var}[\mathrm{E}[f(x)] - z]} = \frac{(\mathrm{E}[f(x)] - z)^2}{\mathrm{Var}[f(x)] + \mathrm{Var}[d] + \mathrm{Var}[e]}$$

where $\mathrm{E}[f(x)]$ and $\mathrm{Var}[f(x)]$ are the emulator expectation and variance respectively; d is the *model discrepancy*, and e is the observational error. The second equality follows from the definition of the *best input approach*.

Several different implausibility measures can be defined in the case where the simulator produces multiple outputs.

8.161 Definition of Term: Inactive input

An inactive input is a *simulator* input that is not an *active input*. Loosely, it is an input that does not influence the output(s) of interest appreciably when it is varied over the required range of input values. More precisely, the values given to inactive inputs in the simulator runs for the *training sample* are ignored when building the *emulator*.

8.162 Definition of Term: MUCM

MUCM stands for 'Managing Uncertainty in Complex Models.' The MUCM project is funded by a research grant from Research Councils UK, in their Basic Technology programme. The objectives of the project are to develop methods for analysing and managing the uncertainty in the outputs produced by *simulators* of physical systems, and as far as possible to show how these methods can be used practically and simply in a wide variety of application areas.

More information about MUCM can be found on the MUCM website.

8.163 Definition of Term: Model Based Design

Several toolkit operations call for a set of points to be specified in the *simulator's* space of inputs, and the choice of such a set is called a design. For instance, to create an *emulator* of the simulator we need a set of points, the *training sample*, at which the simulator is to be run to provide data in order to build the emulator. The question can then be posed, what constitutes a good, or an optimal, design? We cannot answer without having some idea of what the results might be of different choices of design, and in general the results are not known at the time of creating the design – for instance, when choosing a training sample we do not know what the simulator outputs will be at the design points, and so do not know how good the resulting emulator will be. General-purpose designs rely on qualitative generic features of the problem and aim to be good for a wide variety of situations.

In general we should only pick an optimal design, however, if we can specify (a) an optimality criterion (see the alternatives page *AltOptimalCriteria* for some commonly used criteria) and (b) a model for the results (e.g. simulator outputs) that we expect to observe from any design. If one has a reliable model for a prospective emulator, perhaps derived from knowledge of the physical process being modelled by the simulator, then an optimal design may give a more focused experiment. For these reasons we may use the terms "model-based design" which is short for "model based optimal design" .

For optimal design the choice of design points becomes an optimisation problem; e.g. minimise, over the choice of the design, some special criterion. Since the objective is very often to produce an emulator that is as accurate as possible, the optimality criteria are typically phrased in terms of posterior variances (in the case of a fully *Bayesian* emulator, or adjusted variances in the case of a *Bayes linear* emulator).

There are related areas where choice of sites at which to observe or evaluate a function are important. Optimisation is itself such an area and one can also mention search problems involving "queries" in computer science, numerical integration and approximation theory.

8.164 Definition of Term: Model Discrepancy

No matter how careful a particular model of a real system has been formulated, there will always be a difference between reality, represented by the system value y, and the *simulator* output $f(x)$ for any valid input x. The difference $d = y - f(x^+)$, where x^+ is the *best input*, is referred to as the model discrepancy, which should be incorporated into our analysis in order to make valid statements about the real system. In particular, model discrepancy is vital for procedures such as *calibration*, history matching and prediction for the real system.

8.165 Definition of Term: Multilevel Emulation

Multilevel emulation (or *multiscale emulation*) is the application of *emulation* methodology to problems where we have two or more versions of the same *simulator* that produce outputs at different levels of accuracy or resolution. Typically, the lower-accuracy simulators are less expensive to evaluate than the high accuracy models. Multilevel emulation seeks to supplement the restricted amount of information on the most-accurate model, with information gained from larger numbers of evaluations of lower-accuracy simulators to improve the performance of the final emulator.

8.166 Definition of Term: Multivariate Gaussian process

The *univariate Gaussian process* (GP) is a probability distribution for a function, in which the joint distribution of any set of points on that function is multivariate normal. In MUCM it is used to develop an *emulator* for the output of a *simulator*, regarding the simulator output as a function $f(x)$ of its input(s) x.

Most simulators in practice produce multiple outputs (e.g. temperature, pressure, wind speed, . . .) for any given input configuration. The multivariate GP is an extension of the univariate GP to the case where $f(x)$ is not a scalar but a vector. If the simulator has r outputs then $f(x)$ is $r \times 1$.

Formally, the multivariate GP is a probability model over functions with multivariate values. It is characterised by a mean function $m(\cdot) = \textrm{E}[f(\cdot)]$ and a covariance function $v(\cdot,\cdot) = \textrm{Cov}[f(\cdot), f(\cdot)]$. Under this model, the function evaluated at a single input :math:‘ x‘ has a multivariate normal distribution $f(x) \sim N\left(m(x), v(x,x)\right)$, where:

- $m(x)$ is the $r \times 1$ mean vector of $f(x)$ and
- $v(x,x)$ is the $r \times r$ covariance matrix of $f(x)$.

Furthermore, the joint distribution of $f(x)$ and $f(x')$ is also multivariate normal:

$$\left(\begin{array}{c} f(x) \\ f(x') \end{array} \right) \sim N \left(\left(\begin{array}{c} m(x) \\ m(x') \end{array} \right), \left(\begin{array}{cc} v(x,x) & v(x,x') \\ v(x',x) & v(x',x') \end{array} \right) \right)$$

so that $v(x,x')$ is the $r \times r$ matrix of covariances between elements of $f(x)$ and $f(x')$. In general, the multivariate GP is characterised by the fact that if we stack the vectors $f(x_1), f(x_2), \ldots, f(x_n)$ at an arbitrary set of n outputs $D = (x_1, \ldots, x_n)$ into a vector of rn elements, then this has a multivariate normal distribution. The only formal constraint on the covariance function $v(.,.)$ is that the $rn \times rn$ covariance matrix for any arbitrary set D should always be non-negative definite.

The general multivariate GP is therefore very flexible. However, a special case which has the following more restrictive form of covariance function can be very useful:

$$v(\cdot,\cdot) = \Sigma c(\cdot,\cdot),$$

where Σ is a covariance matrix between outputs and $c(\cdot,\cdot)$ is a correlation function between input points.

This restriction implies a kind of *separability* between inputs and outputs. With a covariance function of this form the multivariate GP has an important property, which is exploited in the procedure for emulating multiple outputs *ProcBuildMultiOutputGPSep*. One aspect of this separable property is that when we stack the vectors $f(x_1),\ldots,f(x_n)$ into an $rn \times 1$ vector its covariance matrix has the Kronecker product form $\Sigma \otimes c(D,D)$, where $c(D,D)$ is the between inputs correlation matrix with elements $c(x_i, x_{i'})$ defined according to the notational conventions in the *notation* page.

Another aspect is that if instead of stacking the output vectors into a long vector we form instead the $r \times n$ matrix $f(D)$ (again following the conventions in the *notation* page) then $f(D)$ has a matrix-variate normal distribution with mean matrix $m(D)$, between-rows covariance matrix Σ and between-columns covariance matrix $c(D,D)$.

8.166.1 References

Information on matrix variate distributions can be found in:

- Matrix variate distributions, Arjun K. Gupta, D. K. Nagar, CRC Press, 1999

8.167 Definition of Term: Multivariate t-process

The *univariate t-process* is a probability distribution for a function, in which the joint distribution of any set of points on that function is multivariate t. In MUCM it arises in the fully *Bayesian* approach as the underlying distribution (after integrating out a variance *hyperparameter*) of an *emulator* for the output of a *simulator*, regarding the simulator output as a function $f(x)$ of its input(s) x; see the procedure for building a Gaussian process emulator for the core problem (*ProcBuildCoreGP*). The t-process generalises the *Gaussian process* (GP) in the same way that a t distribution generalises the normal (or Gaussian) distribution.

Most simulators in practice produce multiple outputs (e.g. {temperature, pressure, wind speed, …}) for any given input configuration. If the simulator has r outputs then $f(x)$ is $r \times 1$. In this context, an emulator may be based on the *multivariate GP* or a multivariate t-process.

Formally, the multivariate t-process is a probability model over functions with multivariate values. It is characterised by a degrees of freedom b, a mean function $m(\cdot) = \mathrm{E}[f(\cdot)]$ and a covariance function $v(\cdot,\cdot) = \mathrm{Cov}[f(\cdot), f(\cdot)]$. Under this model, the function evaluated at a single input x has a multivariate t distribution with b degrees of freedom, where:

- $m(x)$ is the $r \times 1$ mean vector of $f(x)$ and
- $v(x,x)$ is the $r \times r$ scale matrix of $f(x)$.

Furthermore, if we stack the vectors $f(x_1), f(x_2), \cdots, f(x_n)$ at an arbitrary set of n outputs $D = (x_1, \cdots, x_n)$ into a vector of rn elements, then this also has a multivariate t distribution.

The multivariate t-process usually arises when we use a multivariate GP model with a *separable* covariance. We therefore shall assume unless specified otherwise that a multivariate t-process also has a *separable* covariance, that is

$$v(\cdot,\cdot) = \Sigma c(\cdot,\cdot)$$

where Σ is a covariance matrix between outputs and $c(\cdot,\cdot)$ is a correlation function between input points.

With a covariance function of this form the multivariate t-process has an important property. If instead of stacking the output vectors into a long vector we form instead the $r \times n$ matrix $f(D)$ (following the conventions in the *notation* page) then $f(D)$ has a matrix-variate t distribution with b degrees of freedom, mean matrix $m(D)$, between-rows covariance matrix Σ and between-columns covariance matrix $c(D,D)$.

8.167.1 References

Information on matrix variate distributions can be found in:

- Matrix variate distributions, Arjun K. Gupta, D. K. Nagar, CRC Press, 1999

8.168 Definition of Term: Nugget

A covariance function $v(x,x')$ expresses the covariance between the outputs of a *simulator* at input configurations x and x'. When $x = x'$, $v(x,x)$ is the variance of the output at input x. A nugget is an additional component of variance when $x = x'$. Technically this results in the covariance function being a discontinuous function of its arguments, because $v(x,x)$ does not equal the limit as x' tends to x of $v(x,x')$.

A nugget may be introduced in a variance function in *MUCM* methods for various reasons. For instance, it may represent random noise in a *stochastic* simulator, or the effects of *inactive inputs* that are not included explicitly in the *emulator*. A small nugget term may also be added for computational reasons when working with a *deterministic* simulator.

8.169 Definition of Term: Principal Component Analysis

Principal Component Analysis (PCA) is a technique used in multivariate data analysis for dimension reduction. Given a sample of vector random variables, the aim is to find a set of orthogonal coordinate axes, known as principal components, with a small number of principal components describing most of the variation in the data. Projecting the data onto the first principal component maximises the variance of the projected points, compared to all other linear projections. Projecting the data onto the second principal component maximises the variance of the projected points, compared to all other linear projections that are orthogonal to the first principal component. Subsequent principal components are defined likewise.

The principal components are obtained using an eigendecomposition of the variance matrix of the data. The eigenvectors give the principal components, and the eigenvalues divided by their sum give the proportion of variation in the data explained by each associated eigenvector (hence the eigenvector with the largest eigenvalue is the first principal component and so on).

If a *simulator* output is multivariate, for example a time series, PCA can be used to reduce the dimension of the output before constructing an *emulator*.

8.170 Definition of Term: Proper (or improper) distribution

A proper distribution is one that integrates (in the case of a probability density function for a continuous random variable) or sums (in the case of a probability mass function for a discrete random variable) to unity. According to probability theory, all probability distributions must have this property.

The concept is relevant in the context of so-called *weak* prior distributions which are claimed to represent (or approximate) a state of ignorance. Such distributions are often improper in the following sense.

Consider a random variable that can take any positive value. One weak prior distribution for such a random variable is the uniform distribution that assigns equal density to all positive values. Such a density function would be given by

$$\pi(x) = k$$

for some positive constant k and for all positive values x of the random variable. However, there is no value of k for which this is a proper distribution. If k is not zero, then the density function integrates to infinity, while if $k = 0$ it integrates to zero. For no value of k does it integrate to one. So such a uniform distribution simply does not exist.

Nevertheless, analysis can often proceed as if this distribution were genuinely proper for some k. That is, by using it as a prior distribution and combining it with the evidence in the data using Bayes' theorem, we will usually obtain a posterior distribution that is proper. The fact that the prior distribution is improper is then unimportant; we regard the resulting posterior distribution as a good approximation to the posterior distribution that we would have obtained from any prior distribution that was actually proper but represented very great prior uncertainty. This use of weak prior distributions is discussed in the definition of the weak prior distribution (*DefWeakPrior*), and is legitimate provided that the supposed approximation really applies. One situation in which it would not is when the improper prior leads to an improper posterior.

Formally, we write the uniform prior as

$$\pi(x) \propto 1\,,$$

using the proportionality symbol to indicate the presence of an unspecified scaling constant k (despite the fact that no such constant can exist). Other improper distributions are specified in like fashion as being proportional to some function, implying a scaling constant which nevertheless cannot exist. An example is another commonly used weak prior distribution for a positive random variable,

$$\pi(x) \propto x^{-1}\,,$$

known as the log-uniform prior.

8.171 Definition of Term: Regularity

The output of a *simulator* is regarded as a function of its inputs. One consideration in building an *emulator* for the simulator is whether the output is expected to be a continuous function of its inputs, and if so whether it is differentiable. Within the *MUCM* toolkit, we refer to the continuity and differentiability properties of a simulator as regularity.

The minimal level of regularity would be a simulator whose output is everywhere a continuous function of the inputs, but is not always differentiable. A function which is differentiable everywhere has greater degree of regularity the more times it is differentiable. The most regular of functions are those that are differentiable infinitely many times.

8.172 Definition of Term: Reification

Perhaps the simplest technique for linking a model to reality is given by the *Best Input* approach. Although very useful and suitable for many applications, often a more detailed approach is required.

A more careful treatment of the link between model and reality, is given by a technique known as Reification. This involves linking the current model f to a Reified model f^+ which would contain all known improvements to the current model. This is often done via an intermediate model f' that represents specific improvements to the current model. The Reified model is subsequently linked to reality.

Emulators are often used to describe each model in this process, as they provide useful representations of the relevant uncertainties involved.

8.173 Definition of Term: Screening

Screening is the process of identifying the *active inputs* that drive a simulator's behaviour, so that only these inputs need to be considered when building and using the emulator. It is common to find that only a small number of inputs to the simulator are relevant to the prediction of the output of interest.

8.174 Definition of Term: Second-order Exchangeability

A sequence of random variables is described as *exchangeable* when our beliefs (in terms of a joint probability distribution) about that collection are unaffected by the order in which they appear.

A less-restrictive specification is that of second-order exchangeability. A sequence of random vectors is described as second-order exchangeable if our beliefs in terms of the mean, variance and covariance for that collection are unaffected by the permutation of the order of the vectors. In other words, this requires that every random vector has the same expectation and variance, and every pair of vectors have the same covariance.

Second-order exchangeability is a concept that is expressed in terms of expectations and variances rather than full probability distributions. It is therefore often exploited in *Bayes linear* analyses.

8.175 Definition of Term: Second-order belief specification

A belief specification for a collection of uncertain quantities is described as *second order* when the beliefs are expressed solely in terms of expectations and variances for every uncertain quantity, and covariances between every pair of quantities. This differs from a fully probabilistic specification which requires a joint probability distribution over all unknown quantities. The second-order specification is the basis for *Bayes Linear* methods.

8.176 Definition of Term: Sensitivity analysis

In general, sensitivity analysis is the process of studying how sensitive the output (here this might be a single variable or a vector of outputs) of a *simulator* is to changes in each of its inputs (and sometimes to combinations of those inputs). There are several different approaches to sensitivity analysis that have been suggested. We can first characterise the methods as local or global.

- *Local sensitivity analysis* consists of seeing how much the output changes when inputs are perturbed by minuscule amounts.
- *Global sensitivity analysis* considers how the output changes when we vary the inputs by larger amounts, reflecting in some sense the range of values of interest for those inputs.

In both approaches, it is usual to specify a *base* set of input values, and to see how the output changes when inputs are varied from from their base values. Essentially, local sensitivity analysis studies the (partial) derivatives of the output with respect to the inputs, evaluated at the base values. Global sensitivity analysis is more complex, because there are a number of different ways to measure the effect on the output, and a number of different ways to define how the inputs are perturbed.

Local sensitivity analysis is very limited because it only looks at the influence of the inputs in a tiny neighbourhood around the base values, and the derivatives are themselves highly sensitive to the scale of measurement of the inputs. For instance, if we decide to measure an input in kilometres instead of metres, then its derivative will be 1000 times larger. Global sensitivity analyses are also influenced by how far we consider varying the inputs. We need a well-defined reason for the choice of ranges, otherwise the sensitivity measures are again arbitrary in the same way as local sensitivity measures respond to an arbitrary scale of measurement.

In the *MUCM* toolkit we generally consider only probabilistic sensitivity analysis, in which the amounts of perturbation are defined by a (joint) probability distribution over the inputs. The usual interpretation of this distribution is as measuring the uncertainty that we have about what the best or "true" values of those inputs should be, in which case the distribution is well defined.

With respect to such a distribution we can define main effects and interactions as follows. Let the model output for input vector X be $f(X)$. Let the i-th input be X_i, and as usual the symbol $\mathrm{E}[\cdot]$ denotes expectation with respect to the probability distribution defined for X.

- The main effect of an input X_i is the function $I_i(x_i) = \mathrm{E}[f(X)\,|\,X_i = x_i] - \mathrm{E}[f(X)]$. So we first take the expected value of the output, averaged over the distribution of all the other inputs conditional on the value of X_i being x_i, then we subtract the overall expected value of the output, averaged over the distribution of all the inputs.
- The interaction between inputs X_i and X_j is the function $I_{\{i,j\}}(x_i, x_j) = \mathrm{E}[f(X)\,|\,X_i = x_i, X_j = x_j] - I_i(x_i) - I_j(x_j) - \mathrm{E}[f(X)]$. This represents deviation in the joint effect of varying the two inputs after subtracting their main effects.

Higher order interactions, involving more than two inputs, are defined analogously.

The main effects and interactions provide a very detailed analysis of how the output responds to the inputs, but we often require simpler, single-figure measures of the sensitivity of the output to individual inputs or combinations of inputs. There are two main ways to do this - *variance based* and *decision based* sensitivity analysis.

8.177 Definition of Term: Separable

An *emulator*'s correlation function specifies the correlation between the outputs of a *simulator* at two different points in its input space. The input space is almost always multi-dimensional, since the simulator will typically have more than one input. Suppose that there are p inputs, so that a point in the input space is a vector x of p elements $x_1, x_2, \ldots, x_p$. Then a correlation function $c(x, x')$ specifies the correlation between simulator outputs at input vectors x and x'. The inputs are said to be separable if the correlation function has the form of a product of one-dimensional correlation functions:

$$c(x, x') = \prod_{i=1}^{p} c_i(x_i, x_i').$$

Specifying the correlation between points that differ in more than one input dimension is potentially a very complex task, particularly because of the constraints involved in creating a valid correlation function. Separability is a property that greatly simplifies this task.

Separability is also used in the context of emulators with multiple outputs. In this case the term 'separable' is typically used to denote separability between the outputs and the inputs, i.e. that all the outputs have the same correlation function $c(x, x')$ (which is often itself separable as defined above). The general covariance then takes the form:

$$\mathrm{Cov}[f_u(x), f_{u'}(x')] = \sigma_{uu'} c(x, x')$$

where u and u' denote two different outputs and $\sigma_{uu'}$ is a covariance between these two outputs.

8.178 Definition of Term: Simulator

A simulator is a representation of some real-world system, usually implemented as a computer program. Given appropriate inputs, the outputs of the simulator are intended to represent specific properties of the real system.

Example: A simulator of a nuclear power station reactor would take as inputs a whole range of quantities describing the detailed design of the reactor and the fuel. Its outputs might include values of heat energy produced by the reactor (perhaps a sustained output level or a time series).

The essence of the *MUCM* toolkit is to describe ways to analyse simulator outputs using the techniques of first building an *emulator*.

8.179 Definition of Term: Single step function

We consider *dynamic simulators* of the form that model the evolution of a *state vector* by iteratively applying a **single step function**. For example, suppose the dynamic simulator takes as inputs an initial state vector w_0, a time series of *forcing inputs* $a_1, \ldots, a_T$, and some constant parameters ϕ, to produce a time series output $w_1, \ldots, w_T$. If the evolution of the time series $w_1, \ldots, w_T$ is defined by the equation $w_t = f(w_{t-1}, a_t, \phi)$, then $f(.)$ is known as the single step function.

8.180 Definition of term: Smoothing kernel

A smoothing kernel is a non-negative real-valued integrable function $\kappa()$ satisfying the following two requirements:

1. $\int_{-\infty}^{+\infty} \kappa(u)\, du$ is finite
2. $\kappa(-u) = \kappa(u)$ for all values of u

In other words, any scalar multiple of a symmetric probability density function constitutes a smoothing kernel.

Smoothing kernels are used in constructing multivariate covariance functions (as discussed in *AltMultivariateCovarianceStructures*), in which case they depend on some hyperparameters. An example of a smoothing kernel in this context is

$$\kappa(x) = \exp\{-0.5 \sum_{i=1}^{p} (x_i/\delta_i)^2\}\,,$$

where p is the length of the vector x. In this case the hyperparameters are $\delta = (\delta_1, ..., \delta_p)$.

8.181 Definition of Term: Smoothness

The output of a *simulator* is regarded as a function of its inputs. One consideration in building an *emulator* for the simulator is whether the output is expected to be a smooth function of the inputs.

Smoothness might be measured formally in a variety of ways. For instance, we might refer to the expected number of times that the output crosses zero in a given range, or the expected number of local maxima or minima it will have in that range (which for a differentiable output corresponds to the number of times its derivative crosses zero). In the toolkit we do not adopt any specific measure of smoothness, but the concept is an important one.

8.182 Definition of Term: Space filling design

Several toolkit operations call for a set of points to be specified in the *simulator* space of inputs, and the choice of such a set is called a design. Space-filling designs are a class of a simple general purpose designs giving sets of points, or *training samples* at which the simulator is run to provide data to build the emulator. They spread points in an approximately uniform way over the design space. They could be any design of the following types.

- a design based on restricted sampling methods.
- a design based on measures of distance between points.
- a design known to have low "discrepancy" interpreted as a formal measure of the closeness to uniformity.
- a design that is a hybrid of, or variation on, these designs.

8.183 Definition of Term: State Vector

The state vector of a *dynamic simulator* comprises a description of the state of the real-world process being modelled by the simulator at any given time point.

Example: A simple simulator of the passage of a drug through a person's body will represent the body as a set of compartments, and its state vector will specify the quantity of the drug in each compartment at a given time point.

8.184 Definition of Term: Stochastic

A *simulator* is referred to as stochastic if running it more than once with a given input configuration will produce randomly different output(s).

Example: A simulator of a hospital admissions system has as inputs a description of the system in terms of number of beds, average time spent in a bed, demand for beds, etc. Its primary output is the number of days in a month in which the hospital is full (all beds occupied). The simulator generates random numbers of patients requiring admission each day, and random lengths of stay for each patient. As a result, the output is random. Running the simulator again with the same inputs (number of beds, etc) will produce different random numbers of patients needing admission and different random lengths of stay, and so the output will also be random.

8.185 Definition of Term: T-process

A t-process is a probability model for an unknown function.

If a function f has argument x, then the value of the function at that argument is $f(x)$. A probability model for such a function must provide a probability distribution for $f(x)$, for any possible argument value x. Furthermore, if we consider a set of possible values for the argument, which we can denote by $x_1, x_2, ..., x_n$, then the probability model must provide a joint probability distribution for the corresponding function values $f(x_1), f(x_2), ..., f(x_n)$.

In *MUCM* methods, an *emulator* is (at least in the fully Bayesian approach) a probability model for the corresponding *simulator*. The simulator is regarded as a function, with the simulator inputs comprising the function's argument and the simulator output(s) comprising the function value.

A t-process is a particular probability model in which the distribution for a single function value is a t distribution (also often called a Student-t distribution), and the joint distribution of a set of function values is multivariate t. The t-process is related to the *Gaussian process* (GP) in the same way that a univariate or multivariate t distribution is related to a univariate or multivariate normal (or Gaussian) distribution. That is, we can think of a t-process as a GP with an uncertain (or random) scaling parameter in its covariance function. Normal and t distributions play a central role in statistical practice by virtue of their mathematical simplicity, and for similar reasons the GP and t-process play a central role in (fully *Bayesian*) MUCM methods.

A t-process is identified by its mean function, its covariance function and a degrees of freedom. If the degrees of freedom is sufficiently large, which will in practice always be the case in MUCM applications, the mean function is E$[f(x)]$, regarded as a function of x, and the covariance function is Cov$[f(x_1), f(x_2)]$, regarded as a function of both x_1 and x_2. Note in particular that the variance of $f(x)$ is the value of the covariance function when both x_1 and x_2 are equal to x.

8.186 Definition of Term: Training sample

In order to build an *emulator* for a *simulator*, we need data comprising the value of the simulator output (or outputs) at each of a set of points in the space of the simulator inputs. Each point in the set is specified as a configuration of input values.

8.187 Definition of Term: Uncertainty analysis

One of the most common tasks for users of *simulators* is to assess the uncertainty in the simulator output(s) that is induced by their uncertainty about the inputs. Input uncertainty is a feature of most applications, since simulators typically require very many input values to be specified and the user will be unsure of what should be the best or correct values for some or all of these.

Uncertainty regarding inputs is characterised by a (joint) probability distribution for their values. This distribution induces a (joint) probability distribution for the output, and uncertainty analysis involves identifying that distribution. Specific tasks might be to compute the mean and variance of the output uncertainty distribution. The mean can be considered as a best estimate for the output in the face of input uncertainty, while the variance measures the amount of output uncertainty. In some applications, it is important to evaluate the probability that the output would lie above or below some threshhold.

A simple way to compute these things in practice is the Monte Carlo method, whereby random configurations of inputs are drawn from their input uncertainty distribution, the model is run for each such configuration, and the set of outputs obtained comprises a random sample from the output distribution. If a sufficiently large sample can be taken, then this allows the uncertainty distribution, mean, variance, probabilities etc., to be evaluated to any desired accuracy. However, this is often impractical because the simulator takes too long to run. Monte Carlo methods will typically require 1,000 to 10,000 runs to achieve accurate estimates of the uncertainty measures, and if a single simulator run takes more than a few seconds the computing time can become prohibitive.

The *MUCM* approach of first building an *emulator* has been developed to enable tasks such as uncertainty analysis to be carried out more efficiently.

Another group of tools that are commonly required are known as *sensitivity analysis*. In particular, the *variance-based* form of sensitivity analysis identifies what proportion of the variance of the uncertainty distribution is attributable to uncertainty in individual inputs, or groups of inputs.

8.188 Definition of Term: Validation

An *emulator* for a *simulator* is a statistical representation of our knowledge about the simulator. In particular, it enables us to predict what outputs the simulator would produce at any given configurations of input values. The process of validation consists of checking whether the actual simulator outputs at those input configurations are consistent with the statistical predictions made by the emulator.

In conjunction with a comparable statistical specification of knowledge about how the simulator relates to the real-world process that it is intended to represent, we can use an emulator to make statistical predictions of the real-world process. These can then also be validated by comparison with the corresponding real-world values.

The validation of a Gaussian process emulator is described in the procedure for validating a Gaussian process emulator (*ProcValidateCoreGP*).

8.189 Definition of Term: Variance-based sensitivity analysis

In variance-based *sensitivity analysis* we consider the effect on the output :math:f(X) of a *simulator<DefSimulator>* as we vary the inputs X, when the variation of those inputs is described by a (joint) probability distribution. This probability distribution can be interpreted as describing uncertainty about the best or true values for the inputs.

We measure the sensitivity to an individual input X_i by the amount of the variance in the output that is attributable to that input. So we begin by considering the variance $\mathrm{Var}[f(X)]$, which represents total uncertainty about the output that is induced by uncertainty about the inputs. In the multi output case this is the variance matrix, which has diagonal elements equivalent to the sensitivity of each output dimension, and cross covariances corresponding to joint sensitivities. This variance also arises as an overall measure of uncertainty in *uncertainty analysis*.

If we were to remove the uncertainty in X_i, then we would expect the uncertainty in $f(X)$ to reduce. The amount of this expected reduction is $V_i = \mathrm{Var}[f(X)] - \mathrm{E}[\mathrm{Var}[f(X) \mid X_i]]$. Notice that the second term in this expression involves first finding the conditional variance of $f(X)$ given that X_i takes some value x_i, which is the uncertainty we would have about the simulator output if we were certain that X_i had that value, so this is the reduced uncertainty in the output. But we are currently uncertain about X_i, so we take an expectation of that conditional variance with respect to our current uncertainty in X_i.

Thus, V_i is defined as the variance-based sensitivity variance for the i-th input. Referring to the definition of the main effect $I_i(x_i)$ in the *sensitivity analysis definition*, it can be shown that V_i is the variance of this main effect. Again for the multi output case this will be a variance matrix, with the main effect being a vector.

We can similarly define the variance-based interaction variance $V_{\{i,j\}}$ of inputs X_i and X_j as the variance of the interaction effect $I_{\{i,j\}}(x_i, x_j)$. When the probability distribution on the inputs is such that they are all independent, then it can be shown that the main effects and interactions (including the higher order interactions that involve three or more inputs) sum to the total variance $\mathrm{Var}[f(X)]$.

The sensitivity variance is often expressed as a proportion of the overall variance $V = \mathrm{Var}[f(X)]$. The ratio $S_i = V_i/V$ is referred to as the sensitivity index for the i-th input, and sensitivity indices for interactions are similarly defined.

Another index of sensitivity for the i-th input that is sometimes used is the total sensitivity index $T_i = \mathrm{E}[\mathrm{Var}[f(X) \mid X_{-i}]]/V$, where X_{-i} means all the inputs *except* X_i. This is the expected proportion of uncertainty in the model output that would be left if we removed the uncertainty in all the inputs except the i-th. In the case of independent inputs, it can be shown that T_i is the sum of the main effect index S_i and the interaction indices for all the interactions involving X_i and one or more other inputs.

8.190 Definition of Term: Weak prior distribution

In *Bayesian* statistics, prior knowledge about a parameter is specified in the form of a probability distribution called its prior distribution (or simply its prior). The use of prior information is a feature of Bayesian statistics, and one which is contentious in the field of statistical inference. Most opponents of the Bayesian approach disagree with the use of prior information. In this context, there has been considerable study of the notion of a prior distribution that represents prior ignorance, or at least a state of very weak prior information, since by using such a prior it may be possible to evade this particular criticism of the Bayesian approach. In this toolkit, we will call such prior distributions “weak priors,” although they may be found answering to many other names in the literature (such as reference priors, noninformative priors, default priors or objective priors).

The use of weak priors has itself been criticised on various grounds. Strict adherents of the Bayesian view argue that genuine prior information invariably exists (i.e. a state of prior ignorance is unrealistic) and that to deny prior information is wasteful. On more pragmatic grounds, it is clear that despite all the research into weak priors there is nothing like a consensus on what is *the* weak prior to use in any situation, and there are numerous competing theories leading to alternative weak prior formulations. On theoretical grounds, others point to logical inconsistencies in any systematic use of weak priors.

In the *MUCM* toolkit, we adopt a Bayesian approach (or a variant known as the *Bayes Linear* approach) and so take the view that prior information should be used. Nevertheless we see a pragmatic value in weak priors. When prior information is genuinely weak relative to the information that may be gained from the data in a statistical analysis, it can be shown that the precise choice of prior distribution has little effect on the statistical results (inferences). Then the use of a weak prior can be justified as being an adequate replacement for spending unnecessary effort on specifying the prior. (And in this context, the existence of alternative weak prior formulations is not a problem because it should not matter which we use.)

Note that conventional weak priors are often *improper*.

CHAPTER 9

mogp_emulator Implementation Details

9.1 The GaussianProcess Class

Implementation of a Gaussian Process Emulator.

This class provides a representation of a Gaussian Process Emulator. It contains methods for fitting the GP to a given set of hyperparameters, computing the negative log marginal likelihood plus prior (so negative log posterior) and its derivatives, and making predictions on unseen data. Note that routines to estimate hyperparameters are not included in the class definition, and are instead provided externally to facilitate implementation of high performance versions.

The required arguments to initialize a GP is a set of training data consisting of the inputs and targets for each of those inputs. These must be numpy arrays whose first axis is of the same length. Targets must be a 1D array, while inputs can be 2D or 1D. If inputs is 1D, then it is assumed that the length of the second axis is unity.

Optional arguments are the particular mean function to use (default is zero mean), the covariance kernel to use (default is the squared exponential covariance), a list of prior distributions for each hyperparameter (default is no prior information on any hyperparameters) and the method for handling the nugget parameter. The nugget is additional "noise" that is added to the diagonal of the covariance kernel as a variance. This nugget can represent uncertainty in the target values themselves, or simply be used to stabilize the numerical inversion of the covariance matrix. The nugget can be fixed (a non-negative float), can be found adaptively (by passing the string `"adaptive"` to make the noise only as large as necessary to successfully invert the covariance matrix), or can be fit as a hyperparameter (by passing the string `"fit"`).

The internal emulator structure involves arrays for the inputs, targets, and hyperparameters. Other useful information are the number of training examples `n`, the number of input parameters `D`, and the number of hyperparameters `n_params`. These parameters can be obtained externally by accessing these attributes

Example:

```
>>> import numpy as np
>>> from mogp_emulator import GaussianProcess
>>> x = np.array([[1., 2., 3.], [4., 5., 6.]])
>>> y = np.array([4., 6.])
>>> gp = GaussianProcess(x, y)
>>> print(gp)
```

(continues on next page)

(continued from previous page)

```
Gaussian Process with 2 training examples and 3 input variables
>>> gp.n
2
>>> gp.D
3
>>> gp.n_params
5
>>> gp.fit(np.zeros(gp.n_params))
>>> x_predict = np.array([[2., 3., 4.], [7., 8., 9.]])
>>> gp.predict(x_predict)
(array([4.74687618, 6.84934016]), array([0.01639298, 1.05374973]),
array([[8.91363045e-05, 7.18827798e-01, 3.74439445e-16],
       [4.64005897e-06, 3.74191346e-02, 1.94917337e-17]]))
```

class mogp_emulator.GaussianProcess.**GaussianProcess**(*inputs*, *targets*, *mean=None*, *kernel=<mogp_emulator.Kernel.SquaredExponential object>*, *priors=None*, *nugget='adaptive'*, *inputdict={}*, *use_patsy=True*)

Implementation of a Gaussian Process Emulator.

This class provides a representation of a Gaussian Process Emulator. It contains methods for fitting the GP to a given set of hyperparameters, computing the negative log marginal likelihood plus prior (so negative log posterior) and its derivatives, and making predictions on unseen data. Note that routines to estimate hyperparameters are not included in the class definition, and are instead provided externally to facilitate implementation of high performance versions.

The required arguments to initialize a GP is a set of training data consisting of the inputs and targets for each of those inputs. These must be numpy arrays whose first axis is of the same length. Targets must be a 1D array, while inputs can be 2D or 1D. If inputs is 1D, then it is assumed that the length of the second axis is unity.

Optional arguments are the particular mean function to use (default is zero mean), the covariance kernel to use (default is the squared exponential covariance), a list of prior distributions for each hyperparameter (default is no prior information on any hyperparameters) and the method for handling the nugget parameter. The nugget is additional "noise" that is added to the diagonal of the covariance kernel as a variance. This nugget can represent uncertainty in the target values themselves, or simply be used to stabilize the numerical inversion of the covariance matrix. The nugget can be fixed (a non-negative float), can be found adaptively (by passing the string `"adaptive"` to make the noise only as large as necessary to successfully invert the covariance matrix), or can be fit as a hyperparameter (by passing the string `"fit"`).

The internal emulator structure involves arrays for the inputs, targets, and hyperparameters. Other useful information are the number of training examples `n`, the number of input parameters `D`, and the number of hyperparameters `n_params`. These parameters can be obtained externally by accessing these attributes

Example:

```
>>> import numpy as np
>>> from mogp_emulator import GaussianProcess
>>> x = np.array([[1., 2., 3.], [4., 5., 6.]])
>>> y = np.array([4., 6.])
>>> gp = GaussianProcess(x, y)
>>> print(gp)
Gaussian Process with 2 training examples and 3 input variables
>>> gp.n
2
>>> gp.D
3
```

(continues on next page)

(continued from previous page)

```
>>> gp.n_params
5
>>> gp.fit(np.zeros(gp.n_params))
>>> x_predict = np.array([[2., 3., 4.], [7., 8., 9.]])
>>> gp.predict(x_predict)
(array([4.74687618, 6.84934016]), array([0.01639298, 1.05374973]),
array([[8.91363045e-05, 7.18827798e-01, 3.74439445e-16],
       [4.64005897e-06, 3.74191346e-02, 1.94917337e-17]]))
```

__init__(*inputs*, *targets*, *mean=None*, *kernel=<mogp_emulator.Kernel.SquaredExponential object>*, *priors=None*, *nugget='adaptive'*, *inputdict={}*, *use_patsy=True*)

Create a new GaussianProcess Emulator

Creates a new GaussianProcess Emulator from either the input data and targets to be fit and optionally a mean function, covariance kernel, and nugget parameter/method.

Required arguments are numpy arrays `inputs` and `targets`, described below. Additional arguments are `mean` to specify the mean function (default is `None` for zero mean), `kernel` to specify the covariance kernel (default is the squared exponential kernel), `priors` to indicate prior distributions on the hyperparameters, and `nugget` to specify how the nugget parameter is handled (see below; default is to fit the nugget adaptively).

`inputs` is a 2D array-like object holding the input data, whose shape is `n` by `D`, where `n` is the number of training examples to be fit and `D` is the number of input variables to each simulation. If `inputs` is a 1D array, it is assumed that `D = 1`.

`targets` is the target data to be fit by the emulator, also held in an array-like object. This must be a 1D array of length `n`.

`prior` must be a list of length `n_params` whose elements are either `Prior`-derived objects or `None`. Each element is used as the prior for the corresponding parameter (with `None` indicating an uninformative prior). Passing the empty list or `None` as this argument (in its entirety) may be used as an abbreviation for a list of `n_params` where all list elements are `None`.

`nugget` controls how additional noise is added to the emulator targets when fitting. This can be specified in several ways. If a string is provided, it can take the values of `"adaptive"` or `"fit"`, which indicate that the nugget will be chosen in the fitting process. `"adaptive"` means that the nugget will be made only as large as necessary to invert the covariance matrix, while `"fit"` means that the nugget will be treated as a hyperparameter to be optimized. Alternatively, a non-negative float can be used to specify a fixed noise level. If no value is specified for the `nugget` parameter, `"adaptive"` is the default.

Parameters

- **inputs** (*ndarray*) – Numpy array holding emulator input parameters. Must be 1D with length `n` or 2D with shape `n` by `D`, where `n` is the number of training examples and `D` is the number of input parameters for each output.
- **targets** (*ndarray*) – Numpy array holding emulator targets. Must be 1D with length `n`
- **mean** (*None or MeanFunction*) – Mean function to be used (optional, default is `None` for a zero mean)
- **kernel** (*Kernel or str*) – Covariance kernel to be used (optional, default is Squared Exponential) Can provide either a `Kernel` object or a string matching the kernel type to be used.
- **priors** (*list or None*) – List of priors to be used. Must be None (default) or an empty list (indicates uninformative priors) or list of length `n_params`. Any parameter

for which you wish to specify an uninformative prior, pass `None`. Number of parameters is the number of parameters in the mean function plus `D + 2` (one correlation length per input plus a covariance scale and a nugget). If the nugget will not be fit, the list can have length `n_params - 1`.

- **nugget** (*float or str*) – Noise to be added to the diagonal, specified as a string or a float. A non-negative float specifies the noise level explicitly, while a string indicates that the nugget will be found via fitting, either as `"adaptive"` or `"fit"` (see above for a description). Default is `"adaptive"`.

Returns New `GaussianProcess` instance

Return type *GaussianProcess*

D

Returns number of inputs for the emulator

Returns Number of inputs for the emulator object

Return type int

fit(*theta*)

Fits the emulator and sets the parameters

Pre-calculates the matrices needed to compute the log-likelihood and its derivatives and make subsequent predictions. This is called any time the hyperparameter values are changed in order to ensure that all the information is needed to evaluate the log-likelihood and its derivatives, which are needed when fitting the optimal hyperparameters.

The method computes the mean function and covariance matrix and inverts the covariance matrix using the method specified by the value of `nugget_type`. The factorized matrix and the product of the inverse with the difference between the targets and the mean are cached for later use, and the negative marginal log-likelihood is also cached. This method has no return value, but it does modify the state of the object.

Parameters **theta** (*ndarray*) – Values of the hyperparameters to use in fitting. Must be a numpy array with length `n_params`

Returns None

get_K_matrix()

Returns current value of the covariance matrix as a numpy array. Does not include the nugget parameter, as this is dependent on how the nugget is fit.

inputs

Returns inputs for the emulator as a numpy array

Returns Emulator inputs, 2D array with shape `(n, D)`

Return type ndarray

logpost_deriv(*theta*)

Calculate the partial derivatives of the negative log-posterior

Calculate the partial derivatives of the negative log-posterior with respect to the hyperparameters. Note that this function is normally used only when fitting the hyperparameters, and it is not needed to make predictions.

During normal use, the `loglike_deriv` method is called after evaluating the `logposterior` method. The implementation takes advantage of this by reusing cached results, as the factorized covariance matrix is expensive to compute and is used by the `logposterior`, `logpost_deriv`, and `logpost_hessian` methods. If the function is evaluated with a different set of parameters than was previously used to set the log-posterior, the method calls `fit` (and subsequently resets the cached information).

Parameters **theta** (*ndarray*) – Value of the hyperparameters. Must be array-like with shape `(n_params,)`

Returns partial derivatives of the negative log-posterior with respect to the hyperparameters (array with shape `(n_params,)`)

Return type ndarray

logpost_hessian(*theta*)

Calculate the Hessian of the negative log-posterior

Calculate the Hessian of the negative log-posterior with respect to the hyperparameters. Note that this function is normally used only when fitting the hyperparameters, and it is not needed to make predictions. It is also used to estimate an appropriate step size when fitting hyperparameters using the lognormal approximation or MCMC sampling.

When used in an optimization routine, the `logpost_hessian` method is called after evaluating the `logposterior` method. The implementation takes advantage of this by storing the inverse of the covariance matrix, which is expensive to compute and is used by the `logposterior` and `logpost_deriv` methods as well. If the function is evaluated with a different set of parameters than was previously used to set the log-posterior, the method calls `fit` to compute the needed information and changes the cached values.

Parameters **theta** (*ndarray*) – Value of the hyperparameters. Must be array-like with shape `(n_params,)`

Returns Hessian of the negative log-posterior (array with shape `(n_params, n_params)`)

Return type ndarray

logposterior(*theta*)

Calculate the negative log-posterior at a particular value of the hyperparameters

Calculate the negative log-posterior for the given set of parameters. Calling this method sets the parameter values and computes the needed inverse matrices in order to evaluate the log-posterior and its derivatives. In addition to returning the log-posterior value, it stores the current value of the hyperparameters and log-posterior in attributes of the object.

Parameters **theta** (*ndarray*) – Value of the hyperparameters. Must be array-like with shape `(n_params,)`

Returns negative log-posterior

Return type float

n

Returns number of training examples for the emulator

Returns Number of training examples for the emulator object

Return type int

n_params

Returns number of hyperparameters

Returns the number of hyperparameters for the emulator. The number depends on the choice of mean function, covariance function, and nugget strategy, and possibly the number of inputs for certain choices of the mean function.

Returns Number of hyperparameters

Return type int

nugget

Returns emulator nugget parameter

Returns current value of the nugget parameter. If the nugget is to be selected adaptively or by fitting the emulator and the nugget has not been fit, returns `None`.

Returns Current nugget value, either a float or `None`

Return type float or None

The `nugget` parameter controls how noise is added to the covariance matrix in order to stabilize the inversion or smooth the emulator predictions. If `nugget` is a non-negative float, then that particular value is used for the nugget. Note that setting this parameter to be zero enforces that the emulator strictly interpolates between points. Alternatively, a string can be provided. A value of `"fit"` means that the nugget is treated as a hyperparameter, and is the last entry in the `theta` array. Alternatively, if `nugget` is set to be `"adaptive"`, the fitting routine will adaptively make the noise parameter as large as is needed to ensure that the emulator can be fit.

Internally, this modifies both the way the nugget is chosen (which can be determined via the `nugget_type` property) and the value itself (the `nugget` property)

Parameters **nugget** (*float or str*) – Noise to be added to the diagonal, specified as a string or a float. A non-negative float specifies the noise level explicitly, while a string indicates that the nugget will be found via fitting, either as `"adaptive"` or `"fit"` (see above for a description).

Returns None

Return type None

nugget_type

Returns method used to select nugget parameter

Returns a string indicating how the nugget parameter is treated, either `"adaptive"`, `"fit"`, or `"fixed"`. This is automatically set when changing the `nugget` property.

Returns Current nugget fitting method

Return type str

predict(*testing*, *unc=True*, *deriv=True*, *include_nugget=True*)

Make a prediction for a set of input vectors for a single set of hyperparameters

Makes predictions for the emulator on a given set of input vectors. The input vectors must be passed as a `(n_predict, D)`, `(n_predict,)` or `(D,)` shaped array-like object, where `n_predict` is the number of different prediction points under consideration and `D` is the number of inputs to the emulator. If the prediction inputs array is 1D and `D == 1` for the GP instance, then the 1D array must have shape `(n_predict,)`. Otherwise, if the array is 1D it must have shape `(D,)`, and the method assumes `n_predict == 1`. The prediction is returned as an `(n_predict, )` shaped numpy array as the first return value from the method.

Optionally, the emulator can also calculate the variances in the predictions and the derivatives with respect to each input parameter. If the uncertainties are computed, they are returned as the second output from the method as an `(n_predict,)` shaped numpy array. If the derivatives are computed, they are returned as the third output from the method as an `(n_predict, D)` shaped numpy array.

The final input to the method determines if the predictive variance should include the nugget or not. For situations where the nugget represents observational error and predictions are estimating the true underlying function, this should be set to `False`. However, most other cases should probably include the nugget, as the emulator is using it to represent some of the uncertainty in the underlying simulator, so the default value is `True`.

Parameters

- **testing** (*ndarray*) – Array-like object holding the points where predictions will be made. Must have shape `(n_predict, D)` or `(D,)` (for a single prediction)
- **unc** (*bool*) – (optional) Flag indicating if the uncertainties are to be computed. If `False` the method returns `None` in place of the uncertainty array. Default value is `True`.
- **deriv** (*bool*) – (optional) Flag indicating if the derivatives are to be computed. If `False` the method returns `None` in place of the derivative array. Default value is `True`.
- **include_nugget** (*bool*) – (optional) Flag indicating if the nugget should be included in the predictive variance. Only relevant if `unc = True`. Default is `True`.

Returns Tuple of numpy arrays holding the predictions, uncertainties, and derivatives, respectively. Predictions and uncertainties have shape `(n_predict,)` while the derivatives have shape `(n_predict, D)`. If the `unc` or `deriv` flags are set to `False`, then those arrays are replaced by `None`.

Return type tuple

priors

The current list priors used in computing the log posterior

To set the priors, must be a list or `None`. Entries can be `None` or a subclass of `Prior`. `None` indicates weak prior information. An empty list or `None` means all uninformative priors. Otherwise list should have the same length as the number of hyperparameters, or alternatively can be one shorter than the number of hyperparameters if `nugget_type` is `"adaptive"` or `"fixed"` meaning that the nugget hyperparameter is not fit but is instead fixed or found adaptively. If the nugget hyperparameter is not fit, the prior for the nugget will automatically be set to `None` even if a distribution is provided.

targets

Returns targets for the emulator as a numpy array

Returns Emulator targets, 1D array with shape `(n,)`

Return type ndarray

theta

Returns emulator hyperparameters

Returns current hyperparameters for the emulator as a numpy array if they have been fit. If no parameters have been fit, returns `None`. Note that the number of parameters depends on the mean function, so the length of this array will vary across instances.

Returns Current parameter values (numpy array of length `n_params`), or `None` if the parameters have not been fit.

Return type ndarray or None

When set, pre-calculates the matrices needed to compute the log-likelihood and its derivatives and make subsequent predictions. This is called any time the hyperparameter values are changed in order to ensure that all the information is needed to evaluate the log-likelihood and its derivatives, which are needed when fitting the optimal hyperparameters.

The method computes the mean function and covariance matrix and inverts the covariance matrix using the method specified by the value of `nugget`. The factorized matrix and the product of the inverse with the difference between the targets and the mean are cached for later use, and the negative marginal log-likelihood is also cached. This method has no return value, but it does modify the state of the object.

Parameters **theta** (*ndarray*) – Values of the hyperparameters to use in fitting. Must be a numpy array with length `n_params`

9.2 The `PredictResult` Class

class mogp_emulator.GaussianProcess.**PredictResult**

Prediction results object

Dictionary-like object containing mean, uncertainty (variance), and derivatives with respect to the inputs of an emulator prediction. Values can be accessed like a dictionary with keys `'mean'`, `'unc'`, and `'deriv'` (or indices 0, 1, and 2 for the mean, uncertainty, and derivative for backwards compatability), or using attributes (`p.mean` if `p` is an instance of `PredictResult`). Also supports iteration and unpacking with the ordering `(mean, unc, deriv)` to be consistent with indexing behavior.

Code is mostly based on scipy's `OptimizeResult` class, with some additional code to support iteration and integer indexing.

Variables

- **mean** – Predicted mean for each input point. Numpy array with shape `(n_predict,)`
- **unc** – Predicted variance for each input point. Numpy array with shape `(n_predict,)`
- **deriv** – Predicted derivative with respect to the inputs for each input point. Numpy array with shape `(n_predict, D)`

9.3 The `MultiOutputGP` Class

Implementation of a multiple-output Gaussian Process Emulator.

Essentially a parallelized wrapper for the predict method. To fit in parallel, use the fit_GP_MAP routine

Required arguments are `inputs` and `targets`, both of which must be numpy arrays. `inputs` can be 1D or 2D (if 1D, assumes second axis has length 1). `targets` can be 1D or 2D (if 2D, assumes a single emulator and the first axis has length 1).

Optional arguments specify how each individual emulator is constructed, including the mean function, kernel, priors, and how to handle the nugget. Each argument can take values allowed by the base `GaussianProcess` class, in which case all emulators are assumed to use the same value. Any of these arguments can alternatively be a list of values with length matching the number of emulators to set those values individually.

Additional keyword arguments include `inputdict`, and `use_patsy`, which control how strings are parsed to mean functions, if using.

class mogp_emulator.MultiOutputGP.**MultiOutputGP**(*inputs*, *targets*, *mean=None*, *kernel=<mogp_emulator.Kernel.SquaredExponential object>*, *priors=None*, *nugget='adaptive'*, *inputdict={}*, *use_patsy=True*)

Implementation of a multiple-output Gaussian Process Emulator.

Essentially a parallelized wrapper for the predict method. To fit in parallel, use the fit_GP_MAP routine

Required arguments are `inputs` and `targets`, both of which must be numpy arrays. `inputs` can be 1D or 2D (if 1D, assumes second axis has length 1). `targets` can be 1D or 2D (if 2D, assumes a single emulator and the first axis has length 1).

Optional arguments specify how each individual emulator is constructed, including the mean function, kernel, priors, and how to handle the nugget. Each argument can take values allowed by the base `GaussianProcess` class, in which case all emulators are assumed to use the same value. Any of these arguments can alternatively be a list of values with length matching the number of emulators to set those values individually.

Additional keyword arguments include `inputdict`, and `use_patsy`, which control how strings are parsed to mean functions, if using.

`__init__`(*inputs*, *targets*, *mean=None*, *kernel=<mogp_emulator.Kernel.SquaredExponential object>*, *priors=None*, *nugget='adaptive'*, *inputdict={}*, *use_patsy=True*)
Create a new multi-output GP Emulator

`predict`(*testing*, *unc=True*, *deriv=True*, *include_nugget=True*, *processes=None*)
Make a prediction for a set of input vectors

Makes predictions for each of the emulators on a given set of input vectors. The input vectors must be passed as a `(n_predict, D)` or `(D,)` shaped array-like object, where `n_predict` is the number of different prediction points under consideration and `D` is the number of inputs to the emulator. If the prediction inputs array has shape `(D,)`, then the method assumes `n_predict == 1`. The prediction points are passed to each emulator and the predictions are collected into an `(n_emulators, n_predict)` shaped numpy array as the first return value from the method.

Optionally, the emulator can also calculate the uncertainties in the predictions (as a variance) and the derivatives with respect to each input parameter. If the uncertainties are computed, they are returned as the second output from the method as an `(n_emulators, n_predict)` shaped numpy array. If the derivatives are computed, they are returned as the third output from the method as an `(n_emulators, n_predict, D)` shaped numpy array. Finally, if uncertainties are computed, the `include_nugget` flag determines if the uncertainties should include the nugget. By default, this is set to `True`.

As with the fitting, this computation can be done independently for each emulator and thus can be done in parallel.

Parameters

- **`testing`** (*`ndarray`*) – Array-like object holding the points where predictions will be made. Must have shape `(n_predict, D)` or `(D,)` (for a single prediction)
- **`unc`** (*`bool`*) – (optional) Flag indicating if the uncertainties are to be computed. If `False` the method returns `None` in place of the uncertainty array. Default value is `True`.
- **`deriv`** (*`bool`*) – (optional) Flag indicating if the derivatives are to be computed. If `False` the method returns `None` in place of the derivative array. Default value is `True`.
- **`include_nugget`** (*`bool`*) – (optional) Flag indicating if the nugget should be included in the predictive variance. Only relevant if `unc = True`. Default is `True`.
- **`processes`** (*`int or None`*) – (optional) Number of processes to use when making the predictions. Must be a positive integer or `None` to use the number of processors on the computer (default is `None`)

Returns Tuple of numpy arrays holding the predictions, uncertainties, and derivatives, respectively. Predictions and uncertainties have shape `(n_emulators, n_predict)` while the derivatives have shape `(n_emulators, n_predict, D)`. If the `do_unc` or `do_deriv` flags are set to `False`, then those arrays are replaced by `None`.

Return type tuple

9.4 The `fitting` Module

`mogp_emulator.fitting.`**`fit_GP_MAP`**(**args*, *n_tries=15*, *theta0=None*, *method='L-BFGS-B'*, ***kwargs*)
Fit one or more Gaussian Processes by attempting to minimize the negative log-posterior

Fits the hyperparameters of one or more Gaussian Processes by attempting to minimize the negative log-posterior multiple times from a given starting location and using a particular minimization method. The best result found among all of the attempts is returned, unless all attempts to fit the parameters result in an error (see below).

The arguments to the method can either be an existing `GaussianProcess` or `MultiOutputGP` instance, or a list of arguments to be passed to the `__init__` method of `GaussianProcess` or `MultiOutputGP` if more than one output is detected. Keyword arguments for creating a new `GaussianProcess` or `MultiOutputGP` object can either be passed as part of the `*args` list or as keywords (if present in `**kwargs`, they will be extracted and passed separately to the `__init__` method).

If the method encounters an overflow (this can result because the parameter values stored are the logarithm of the actual hyperparameters to enforce positivity) or a linear algebra error (occurs when the covariance matrix cannot be inverted, even with additional noise added along the diagonal if adaptive noise was selected), the iteration is skipped. If all attempts to find optimal hyperparameters result in an error, then the method raises an exception.

The `theta0` parameter is the point at which the first iteration will start. If more than one attempt is made, subsequent attempts will use random starting points. If you are fitting Multiple Outputs, then this argument can take any of the following forms: (1) None (random start points for all emulators), (2) a list of numpy arrays or `NoneTypes` with length `n_emulators`, (3) a numpy array of shape `(n_params,)` or `(n_emulators, n_params)` which with either use the same start point for all emulators or the specified start point for all emulators. Note that if you us a numpy array, all emulators must have the same number of parameters, while using a list allows more flexibility.

The user can specify the details of the minimization method, using any of the gradient-based optimizers available in `scipy.optimize.minimize`. Any additional parameters beyond the method specification can be passed as keyword arguments.

The function returns a fit `GaussianProcess` or `MultiOutputGP` instance, either the original one passed to the function, or the new one created from the included arguments.

Parameters

- **`*args`** – Either a single `GaussianProcess` or `MultiOutputGP` instance, or arguments to be passed to the `__init__` method when creating a new `GaussianProcess` or `MultiOutputGP` instance.
- **`n_tries`** (*`int`*) – Number of attempts to minimize the negative log-posterior function. Must be a positive integer (optional, default is 15)
- **`theta0`** (*`None or ndarray`*) – Initial starting point for the first iteration. If present, must be array-like with shape `(n_params,)` based on the specific `GaussianProcess` being fit. If a `MultiOutputGP` is being fit it must be a list of length `n_emulators` with each entry as either `None` or a numpy array of shape `(n_params,)`, or a numpy array with shape `(n_emulators, n_params)` (note that if the various emulators have different numbers of parameters, the numpy array option will not work). If `None` is given, then a random value is chosen. (Default is `None`)
- **`method`** (*`str`*) – Minimization method to be used. Can be any gradient-based optimization method available in `scipy.optimize.minimize`. (Default is `'L-BFGS-B'`)
- **`**kwargs`** – Additional keyword arguments to be passed to `GaussianProcess.__init__`, `MultiOutputGP.__init__`, or the minimization routine. Relevant parameters for the GP classes are automatically split out from those used in the minimization function. See available parameters in the corresponding functions for details.

Returns Fit GP or Multi-Output GP instance

Return type *GaussianProcess* or *MultiOutputGP*

9.5 The `MeanFunction` Module

MeanFunction Module

The MeanFunction module contains classes used for constructing mean functions for GP emulators. A base `MeanBase` class is provided, which implements basic operations to combine fixed functions and fitting parameters. The basic operations `f1 + f2`, `f1*f2`, `f1**f2` and `f1(f2)` are available, though not all possible combinations will make sense. Particular cases where combinations do not make sense often use classes that represent free fitting parameters, or if you attempt to raise a mean function to a power that is not independent of the inputs. These operations will create new derived classes `MeanSum`, `MeanProduct`, `MeanPower`, and `MeanComposite`, from which more complex regression functions can be formed. The derived sum, product, power, and composite mean classes call the necessary methods to compute the function and derivatives from the more basic classes and then combine them using sum, product, power, and chain rules for function evaluation and derivatives.

The basic building blocks are fixed mean functions, derived from `FixedMean`, and free parameters, represented by the `Coefficient` class. Incuded fixed functions include `ConstantMean` and `LinearMean`. Additional derived `FixedMean` functions can be created by initializing a new `FixedMean` instance where the user provides a fixed function and its derivative, and these can be combined to form arbitrarily complex mean functions. Future improvements will extend the number of pre-defined function options.

One implementation note: `CompositeMean` does not implement the Hessian, as computing this requires mixed partials involving inputs and parameters that are not normally implemented. If a composite mean is required with a Hessian Computation, the user must implement this.

Additionally, note that given mean function may have a number of parameters that depends on the shape of the input. Since the mean function does not store input, but rather provides a way to collate functions and derivatives together in a single object, the number of parameters can vary based on the inputs. This is particularly true for the provided `PolynomialMean` class, which fits a polynomial function of a fixed degree to each input parameter. Thus, the number of parameters depends on the input shape.

In addition to manually creating a mean function by composing fixed functions and fitting parameters, a `MeanBase` subclass can be created by using the `MeanFunction` function. `MeanFunction` is a functional interface for creating `MeanBase` subclasses from a string formula. The formula langauge supports the operations described above as expected (see below for some examples), with the option to first parse the formula using the Python library `patsy` before converting the terms to the respective subclasses of `MeanBase`. Formulas specify input variables using either `x[i]` or `inputs[i]` to represent the dependent variables, and can explicitly include a leading `"y ="` or `"y ~"` (which will be ignored). Optionally, named variables can be mapped to input dimensions by providing a dictionary mapping strings to integer indices. Any other variables in the formula will be assumed to be fitting coefficients. Note that the formula parser does not make any effort to simplify expressions (such as having identical terms or a term with redundant fitting parameters), so it is up to the user to get things correct. Converting a mean function instance to a string can be very helpful in determining if the parsing led to any problems, see below.

Example:

```
>>> from mogp_emulator.MeanFunction import Coefficient, LinearMean, MeanFunction
>>> mf1 = Coefficient() + Coefficient()*LinearMean()
>>> print(mf1)
c + c*x[0]
>>> mf2 = LinearMean(1)*LinearMean(2)
>>> print(mf2)
x[1]*x[2]
>>> mf3 = mf1(mf2)
>>> print(mf3)
c + c*x[1]*x[2]
>>> mf4 = Coefficient()*LinearMean()**2
>>> print(mf4)
c*x[0]^2
```

(continues on next page)

(continued from previous page)

```
>>> mf5 = MeanFunction("x[0]")
>>> print(mf5)
c + c*x[0]
>>> mf6 = MeanFunction("y = a + b*x[0]", use_patsy=False)
>>> print(mf6)
c + c*x[0]
>>> mf7 = MeanFunction("a*b", {"a": 0, "b": 1})
>>> print(mf7)
c + c*x[0] + c*x[1] + c*x[0]*x[1]
```

`mogp_emulator.MeanFunction.`**`MeanFunction`**(*formula*, *inputdict={}*, *use_patsy=True*)

Create a mean function from a formula

This is the functional interface to creating a mean function from a string formula. This method takes a string as an input, an optional dictionary that map strings to integer indices in the input data, and an optional boolean flag that indicates if the user would like to have the formula parsed with patsy before being converted to a mean function.

The string formulas can be specified in several ways. The formula LHS is implicitly always `"y = "` or `"y ~ "`, though these can be explicitly provided as well. The RHS may contain a set of terms containing the add, multiply, power, and call operations much in the same way that the operations would be entered as regular python code. Parentheses are used to indicated prececence as well as the call operation, and square brackets indicate an indexing operation on the inputs. Inputs may be specified as either a string such as `"x[0]"`, `"inputs[0]"`, or a string that can be mapped to an integer index with the optional dictionary passed to the function. Any strings not representing operations or inputs as described above are interpreted as follows: if the string can be converted into a number, then it is interpreted as a `ConstantMean` fixed mean function object; otherwise it is assumed to represent a fitting coefficient. Note that this means many characters that do not represent operations within this mean function language but would not normally be considered as python variables will nonetheless be converted into fitting coefficients – it is up to the user to get this right.

Expressions that are repeated or redundant will not be simplified, so the user should take care that the provided expression is sensible as a mean function and will not cause problems when fitting.

Additional special cases to be aware of:

- `call` cannot be used as a variable name, if this is parsed as a token an exception will be raised.
- `I` is the identity operator, it simply returns the given value. It is useful if you wish to use patsy to evaluate a formula but protect a part of the string formula from being expanded based on the rules in patsy. If `I` is encountered in any other context, an exception will be raised.

Examples:

```
>>> from mogp_emulator.MeanFunction import MeanFunction
>>> mf1 = MeanFunction("x[0]")
>>> print(mf1)
c + c*x[0]
>>> mf2 = MeanFunction("y = a + b*x[0]", use_patsy=False)
>>> print(mf2)
c + c*x[0]
>>> mf3 = MeanFunction("a*b", {"a": 0, "b": 1})
>>> print(mf3)
c + c*x[0] + c*x[1] + c*x[0]*x[1]
```

Parameters

- **`formula`** (*`str`*) – string representing the desired mean function formula

- **inputdict** (*dict*) – dictionary used to map variables to input indices. Maps strings to integer indices (must be non-negative). Optional, default is {}.
- **use_patsy** (*bool*) – Boolean flag indicating if the string is to be parsed using patsy library. Optional, default is True. If patsy is not installed, the basic string parser will be used.

Returns New subclass of MeanBase implementing the given formula

Return type subclass of MeanBase (exact type will depend on the formula that is provided)

class mogp_emulator.MeanFunction.**MeanBase**

Base mean function class

The base class for the mean function implementation includes code for checking inputs and implements sum, product, power, and composition methods to allow more complicated functions to be built up from fixed functions and fitting coefficients. Subclasses need to implement the following methods:

- get_n_params which returns the number of parameters for a given input size. This is usually a constant, but can be more complicated (such as the provided PolynomialMean class)
- mean_f computes the mean function for the provided inputs and parameters
- mean_deriv computes the derivative with respect to the parameters
- mean_hessian computes the hessian with respect to the parameters
- mean_inputderiv computes the derivate with respect to the inputs

The base class does not have any attributes, but subclasses will usually have some attributes that must be set and so are likely to need a __init__ method.

get_n_params (*x*)

Determine the number of parameters

Returns the number of parameters for the mean function, which possibly depends on x.

Parameters **x** (*ndarray*) – Input array

Returns number of parameters

Return type int

mean_deriv (*x*, *params*)

Returns value of mean function derivative wrt the parameters

Method to compute the value of the mean function derivative with respect to the parameters for the inputs and parameters provided. Shapes of x and params must be consistent based on the return value of the get_n_params method. Returns a numpy array of shape (n_params, x.shape[0]) holding the value of the mean function derivative with respect to each parameter (first axis) for each input point (second axis).

Parameters

- **x** (*ndarray*) – Inputs, must be a 1D or 2D numpy array (if 1D a second dimension will be added)
- **params** (*ndarray*) – Parameters, must be a 1D numpy array (of more than 1D will be flattened) and have the same length as the number of parameters required for the provided input

Returns Value of mean function derivative with respect to the parameters evaluated at all input points, numpy array of shape (n_params, x.shape[0])

Return type ndarray

mean_f (*x*, *params*)

Returns value of mean function

Method to compute the value of the mean function for the inputs and parameters provided. Shapes of `x` and `params` must be consistent based on the return value of the `get_n_params` method. Returns a numpy array of shape `(x.shape[0],)` holding the value of the mean function for each input point.

Parameters

- **x** (*ndarray*) – Inputs, must be a 1D or 2D numpy array (if 1D a second dimension will be added)
- **params** (*ndarray*) – Parameters, must be a 1D numpy array (of more than 1D will be flattened) and have the same length as the number of parameters required for the provided input

Returns Value of mean function evaluated at all input points, numpy array of shape `(x.shape[0],)`

Return type ndarray

mean_hessian (*x*, *params*)

Returns value of mean function Hessian wrt the parameters

Method to compute the value of the mean function Hessian with respect to the parameters for the inputs and parameters provided. Shapes of `x` and `params` must be consistent based on the return value of the `get_n_params` method. Returns a numpy array of shape `(n_params, n_params, x.shape[0])` holding the value of the mean function second derivaties with respect to each parameter pair (first twp axes) for each input point (last axis).

Parameters

- **x** (*ndarray*) – Inputs, must be a 1D or 2D numpy array (if 1D a second dimension will be added)
- **params** (*ndarray*) – Parameters, must be a 1D numpy array (of more than 1D will be flattened) and have the same length as the number of parameters required for the provided input

Returns Value of mean function Hessian with respect to the parameters evaluated at all input points, numpy array of shape `(n_parmas, n_params, x.shape[0])`

Return type ndarray

mean_inputderiv (*x*, *params*)

Returns value of mean function derivative wrt the inputs

Method to compute the value of the mean function derivative with respect to the inputs for the inputs and parameters provided. Shapes of `x` and `params` must be consistent based on the return value of the `get_n_params` method. Returns a numpy array of shape `(x.shape[1], x.shape[0])` holding the value of the mean function derivative with respect to each input (first axis) for each input point (second axis).

Parameters

- **x** (*ndarray*) – Inputs, must be a 1D or 2D numpy array (if 1D a second dimension will be added)
- **params** (*ndarray*) – Parameters, must be a 1D numpy array (of more than 1D will be flattened) and have the same length as the number of parameters required for the provided input

Returns Value of mean function derivative with respect to the inputs evaluated at all input points, numpy array of shape `(x.shape[1], x.shape[0])`

Return type ndarray

class `mogp_emulator.MeanFunction.`**`MeanSum`**(*f1*, *f2*)

Class representing the sum of two mean functions

This derived class represents the sum of two mean functions, and does the necessary bookkeeping needed to compute the required function and derivatives. The code does not do any checks to confirm that it makes sense to add these particular mean functions – in particular, adding two `Coefficient` classes is the same as having a single one, but the code will not attempt to simplify this so it is up to the user to get it right.

Variables

- **f1** – first `MeanBase` to be added
- **f2** – second `MeanBase` to be added

`get_n_params`(*x*)

Determine the number of parameters

Returns the number of parameters for the mean function, which possibly depends on x.

Parameters **x** (*ndarray*) – Input array

Returns number of parameters

Return type int

`mean_deriv`(*x*, *params*)

Returns value of mean function derivative wrt the parameters

Method to compute the value of the mean function derivative with respect to the parameters for the inputs and parameters provided. Shapes of `x` and `params` must be consistent based on the return value of the `get_n_params` method. Returns a numpy array of shape `(n_params, x.shape[0])` holding the value of the mean function derivative with respect to each parameter (first axis) for each input point (second axis).

For `MeanSum`, this method applies the sum rule to the results of computing the derivative for the individual functions.

Parameters

- **x** (*ndarray*) – Inputs, must be a 1D or 2D numpy array (if 1D a second dimension will be added)
- **params** (*ndarray*) – Parameters, must be a 1D numpy array (of more than 1D will be flattened) and have the same length as the number of parameters required for the provided input

Returns Value of mean function derivative with respect to the parameters evaluated at all input points, numpy array of shape `(n_params, x.shape[0])`

Return type ndarray

`mean_f`(*x*, *params*)

Returns value of mean function

Method to compute the value of the mean function for the inputs and parameters provided. Shapes of `x` and `params` must be consistent based on the return value of the `get_n_params` method. Returns a numpy array of shape `(x.shape[0],)` holding the value of the mean function for each input point.

For `MeanSum`, this method applies the sum rule to the results of computing the mean for the individual functions.

Parameters

- **x** (*ndarray*) – Inputs, must be a 1D or 2D numpy array (if 1D a second dimension will be added)
- **params** (*ndarray*) – Parameters, must be a 1D numpy array (of more than 1D will be flattened) and have the same length as the number of parameters required for the provided input

Returns Value of mean function evaluated at all input points, numpy array of shape `(x.shape[0],)`

Return type ndarray

mean_hessian (*x*, *params*)

Returns value of mean function Hessian wrt the parameters

Method to compute the value of the mean function Hessian with respect to the parameters for the inputs and parameters provided. Shapes of `x` and `params` must be consistent based on the return value of the `get_n_params` method. Returns a numpy array of shape `(n_params, n_params, x.shape[0])` holding the value of the mean function second derivaties with respect to each parameter pair (first twp axes) for each input point (last axis).

For `MeanSum`, this method applies the sum rule to the results of computing the Hessian for the individual functions.

Parameters

- **x** (*ndarray*) – Inputs, must be a 1D or 2D numpy array (if 1D a second dimension will be added)
- **params** (*ndarray*) – Parameters, must be a 1D numpy array (of more than 1D will be flattened) and have the same length as the number of parameters required for the provided input

Returns Value of mean function Hessian with respect to the parameters evaluated at all input points, numpy array of shape `(n_parmas, n_params, x.shape[0])`

Return type ndarray

mean_inputderiv (*x*, *params*)

Returns value of mean function derivative wrt the inputs

Method to compute the value of the mean function derivative with respect to the inputs for the inputs and parameters provided. Shapes of `x` and `params` must be consistent based on the return value of the `get_n_params` method. Returns a numpy array of shape `(x.shape[1], x.shape[0])` holding the value of the mean function derivative with respect to each input (first axis) for each input point (second axis).

For `MeanSum`, this method applies the sum rule to the results of computing the derivative for the individual functions.

Parameters

- **x** (*ndarray*) – Inputs, must be a 1D or 2D numpy array (if 1D a second dimension will be added)
- **params** (*ndarray*) – Parameters, must be a 1D numpy array (of more than 1D will be flattened) and have the same length as the number of parameters required for the provided input

Returns Value of mean function derivative with respect to the inputs evaluated at all input points, numpy array of shape `(x.shape[1], x.shape[0])`

Return type ndarray

class mogp_emulator.MeanFunction.**MeanProduct**(*f1*, *f2*)

Class representing the product of two mean functions

This derived class represents the product of two mean functions, and does the necessary bookkeeping needed to compute the required function and derivatives. The code does not do any checks to confirm that it makes sense to multiply these particular mean functions – in particular, multiplying two `Coefficient` classes is the same as having a single one, but the code will not attempt to simplify this so it is up to the user to get it right.

Variables

- **f1** – first `MeanBase` to be multiplied
- **f2** – second `MeanBase` to be multiplied

get_n_params(*x*)

Determine the number of parameters

Returns the number of parameters for the mean function, which possibly depends on x.

Parameters **x** (*ndarray*) – Input array

Returns number of parameters

Return type int

mean_deriv(*x*, *params*)

Returns value of mean function derivative wrt the parameters

Method to compute the value of the mean function derivative with respect to the parameters for the inputs and parameters provided. Shapes of `x` and `params` must be consistent based on the return value of the `get_n_params` method. Returns a numpy array of shape `(n_params, x.shape[0])` holding the value of the mean function derivative with respect to each parameter (first axis) for each input point (second axis).

For `MeanProduct`, this method applies the product rule to the results of computing the derivative for the individual functions.

Parameters

- **x** (*ndarray*) – Inputs, must be a 1D or 2D numpy array (if 1D a second dimension will be added)
- **params** (*ndarray*) – Parameters, must be a 1D numpy array (of more than 1D will be flattened) and have the same length as the number of parameters required for the provided input

Returns Value of mean function derivative with respect to the parameters evaluated at all input points, numpy array of shape `(n_params, x.shape[0])`

Return type ndarray

mean_f(*x*, *params*)

Returns value of mean function

Method to compute the value of the mean function for the inputs and parameters provided. Shapes of `x` and `params` must be consistent based on the return value of the `get_n_params` method. Returns a numpy array of shape `(x.shape[0],)` holding the value of the mean function for each input point.

For `MeanProduct`, this method applies the product rule to the results of computing the mean for the individual functions.

Parameters

- **x** (*ndarray*) – Inputs, must be a 1D or 2D numpy array (if 1D a second dimension will be added)
- **params** (*ndarray*) – Parameters, must be a 1D numpy array (of more than 1D will be flattened) and have the same length as the number of parameters required for the provided input

Returns Value of mean function evaluated at all input points, numpy array of shape `(x.shape[0],)`

Return type ndarray

mean_hessian(*x*, *params*)

Returns value of mean function Hessian wrt the parameters

Method to compute the value of the mean function Hessian with respect to the parameters for the inputs and parameters provided. Shapes of `x` and `params` must be consistent based on the return value of the `get_n_params` method. Returns a numpy array of shape `(n_params, n_params, x.shape[0])` holding the value of the mean function second derivaties with respect to each parameter pair (first twp axes) for each input point (last axis).

For `MeanProduct`, this method applies the product rule to the results of computing the Hessian for the individual functions.

Parameters

- **x** (*ndarray*) – Inputs, must be a 1D or 2D numpy array (if 1D a second dimension will be added)
- **params** (*ndarray*) – Parameters, must be a 1D numpy array (of more than 1D will be flattened) and have the same length as the number of parameters required for the provided input

Returns Value of mean function Hessian with respect to the parameters evaluated at all input points, numpy array of shape `(n_parmas, n_params, x.shape[0])`

Return type ndarray

mean_inputderiv(*x*, *params*)

Returns value of mean function derivative wrt the inputs

Method to compute the value of the mean function derivative with respect to the inputs for the inputs and parameters provided. Shapes of `x` and `params` must be consistent based on the return value of the `get_n_params` method. Returns a numpy array of shape `(x.shape[1], x.shape[0])` holding the value of the mean function derivative with respect to each input (first axis) for each input point (second axis).

For `MeanProduct`, this method applies the product rule to the results of computing the derivative for the individual functions.

Parameters

- **x** (*ndarray*) – Inputs, must be a 1D or 2D numpy array (if 1D a second dimension will be added)
- **params** (*ndarray*) – Parameters, must be a 1D numpy array (of more than 1D will be flattened) and have the same length as the number of parameters required for the provided input

Returns Value of mean function derivative with respect to the inputs evaluated at all input points, numpy array of shape `(x.shape[1], x.shape[0])`

Return type ndarray

class mogp_emulator.MeanFunction.**MeanPower**(*f1*, *f2*)

Class representing a mean function raised to a power

This derived class represents a mean function raised to a power, and does the necessary bookkeeping needed to compute the required function and derivatives. The code requires that the exponent be either a `Coefficient`, `ConstantMean`, `float`, or `int` as the output of the exponent mean function must be independent of the inputs to make sense. If input is a float or int, a `ConstantMean` instance will be created.

Variables

- **f1** – first `MeanBase` to be raised to the given exponent
- **f2** – second `MeanBase` indicating the exponent. Must be a `Coefficient`, `ConstantMean`, or float/int (from which a `ConstantMean` object will be created)

get_n_params(*x*)

Determine the number of parameters

Returns the number of parameters for the mean function, which possibly depends on x.

Parameters **x** (*ndarray*) – Input array

Returns number of parameters

Return type int

mean_deriv(*x*, *params*)

Returns value of mean function derivative wrt the parameters

Method to compute the value of the mean function derivative with respect to the parameters for the inputs and parameters provided. Shapes of `x` and `params` must be consistent based on the return value of the `get_n_params` method. Returns a numpy array of shape `(n_params, x.shape[0])` holding the value of the mean function derivative with respect to each parameter (first axis) for each input point (second axis).

For `MeanPpwer`, this method applies the power rule to the results of computing the derivative for the individual functions.

Parameters

- **x** (*ndarray*) – Inputs, must be a 1D or 2D numpy array (if 1D a second dimension will be added)
- **params** (*ndarray*) – Parameters, must be a 1D numpy array (of more than 1D will be flattened) and have the same length as the number of parameters required for the provided input

Returns Value of mean function derivative with respect to the parameters evaluated at all input points, numpy array of shape `(n_params, x.shape[0])`

Return type ndarray

mean_f(*x*, *params*)

Returns value of mean function

Method to compute the value of the mean function for the inputs and parameters provided. Shapes of `x` and `params` must be consistent based on the return value of the `get_n_params` method. Returns a numpy array of shape `(x.shape[0],)` holding the value of the mean function for each input point.

For `MeanProduct`, this method applies the product rule to the results of computing the mean for the individual functions.

Parameters

- **x** (*ndarray*) – Inputs, must be a 1D or 2D numpy array (if 1D a second dimension will be added)
- **params** (*ndarray*) – Parameters, must be a 1D numpy array (of more than 1D will be flattened) and have the same length as the number of parameters required for the provided input

Returns Value of mean function evaluated at all input points, numpy array of shape `(x.shape[0],)`

Return type ndarray

mean_hessian (*x*, *params*)

Returns value of mean function Hessian wrt the parameters

Method to compute the value of the mean function Hessian with respect to the parameters for the inputs and parameters provided. Shapes of `x` and `params` must be consistent based on the return value of the `get_n_params` method. Returns a numpy array of shape `(n_params, n_params, x.shape[0])` holding the value of the mean function second derivaties with respect to each parameter pair (first twp axes) for each input point (last axis).

For `MeanPower`, this method applies the power rule to the results of computing the Hessian for the individual functions.

Parameters

- **x** (*ndarray*) – Inputs, must be a 1D or 2D numpy array (if 1D a second dimension will be added)
- **params** (*ndarray*) – Parameters, must be a 1D numpy array (of more than 1D will be flattened) and have the same length as the number of parameters required for the provided input

Returns Value of mean function Hessian with respect to the parameters evaluated at all input points, numpy array of shape `(n_parmas, n_params, x.shape[0])`

Return type ndarray

mean_inputderiv (*x*, *params*)

Returns value of mean function derivative wrt the inputs

Method to compute the value of the mean function derivative with respect to the inputs for the inputs and parameters provided. Shapes of `x` and `params` must be consistent based on the return value of the `get_n_params` method. Returns a numpy array of shape `(x.shape[1], x.shape[0])` holding the value of the mean function derivative with respect to each input (first axis) for each input point (second axis).

For `MeanPower`, this method applies the power rule to the results of computing the derivative for the individual functions.

Parameters

- **x** (*ndarray*) – Inputs, must be a 1D or 2D numpy array (if 1D a second dimension will be added)
- **params** (*ndarray*) – Parameters, must be a 1D numpy array (of more than 1D will be flattened) and have the same length as the number of parameters required for the provided input

Returns Value of mean function derivative with respect to the inputs evaluated at all input points, numpy array of shape `(x.shape[1], x.shape[0])`

Return type ndarray

class mogp_emulator.MeanFunction.**MeanComposite**(*f1*, *f2*)

Class representing the composition of two mean functions

This derived class represents the composition of two mean functions, and does the necessary bookkeeping needed to compute the required function and derivatives. The code does not do any checks to confirm that it makes sense to compose these particular mean functions – in particular, applying a `Coefficient` class to another function will simply wipe out the second function. This will not raise an error, but the code will not attempt to alert the user to this so it is up to the user to get it right.

Because the Hessian computation requires mixed partials that are not normally implemented in the `MeanBase` class, the Hessian computation is not currently implemented. If you require Hessian computation for a composite mean function, you must implement it yourself.

Note that since the outer function takes as its input the output of the second function, the outer function can only ever have an index of 0 due to the fixed output shape of a mean function. This will produce an error when attempting to evaluate the function or its derivatives, but will not cause an error when initializing a `MeanComposite` object.

Variables

- **f1** – first `MeanBase` to be applied to the second
- **f2** – second `MeanBase` to be composed as the input to the first

get_n_params(*x*)

Determine the number of parameters

Returns the number of parameters for the mean function, which possibly depends on x.

Parameters **x** (*ndarray*) – Input array

Returns number of parameters

Return type int

mean_deriv(*x*, *params*)

Returns value of mean function derivative wrt the parameters

Method to compute the value of the mean function derivative with respect to the parameters for the inputs and parameters provided. Shapes of `x` and `params` must be consistent based on the return value of the `get_n_params` method. Returns a numpy array of shape `(n_params, x.shape[0])` holding the value of the mean function derivative with respect to each parameter (first axis) for each input point (second axis).

For `MeanComposite`, this method applies the chain rule to the results of computing the derivative for the individual functions.

Parameters

- **x** (*ndarray*) – Inputs, must be a 1D or 2D numpy array (if 1D a second dimension will be added)
- **params** (*ndarray*) – Parameters, must be a 1D numpy array (of more than 1D will be flattened) and have the same length as the number of parameters required for the provided input

Returns Value of mean function derivative with respect to the parameters evaluated at all input points, numpy array of shape `(n_params, x.shape[0])`

Return type ndarray

mean_f(*x*, *params*)

Returns value of mean function

Method to compute the value of the mean function for the inputs and parameters provided. Shapes of `x` and `params` must be consistent based on the return value of the `get_n_params` method. Returns a numpy array of shape `(x.shape[0],)` holding the value of the mean function for each input point.

For `MeanComposite`, this method applies the output of the second function as input to the first function.

Parameters

- **x** (*ndarray*) – Inputs, must be a 1D or 2D numpy array (if 1D a second dimension will be added)
- **params** (*ndarray*) – Parameters, must be a 1D numpy array (of more than 1D will be flattened) and have the same length as the number of parameters required for the provided input

Returns Value of mean function evaluated at all input points, numpy array of shape `(x.shape[0],)`

Return type ndarray

mean_inputderiv(*x*, *params*)

Returns value of mean function derivative wrt the inputs

Method to compute the value of the mean function derivative with respect to the inputs for the inputs and parameters provided. Shapes of `x` and `params` must be consistent based on the return value of the `get_n_params` method. Returns a numpy array of shape `(x.shape[1], x.shape[0])` holding the value of the mean function derivative with respect to each input (first axis) for each input point (second axis).

For `MeanComposite`, this method applies the chain rule to the results of computing the derivative for the individual functions.

Parameters

- **x** (*ndarray*) – Inputs, must be a 1D or 2D numpy array (if 1D a second dimension will be added)
- **params** (*ndarray*) – Parameters, must be a 1D numpy array (of more than 1D will be flattened) and have the same length as the number of parameters required for the provided input

Returns Value of mean function derivative with respect to the inputs evaluated at all input points, numpy array of shape `(x.shape[1], x.shape[0])`

Return type ndarray

class mogp_emulator.MeanFunction.**FixedMean**(*f*, *deriv=None*)

Class representing a fixed mean function with no parameters

Class representing a mean function with a fixed function (and optional derivative) and no fitting parameters. The user must provide these functions when initializing the instance.

Variables

- **f** – fixed mean function, must be callable and take a single argument (the inputs)
- **deriv** – fixed derivative function (optional if no derivatives are needed), must be callable and take a single argument (the inputs)

get_n_params(*x*)

Determine the number of parameters

Returns the number of parameters for the mean function, which possibly depends on x. For a `FixedMean` class, this is zero.

Parameters **x** (*ndarray*) – Input array

Returns number of parameters

Return type int

mean_deriv (*x*, *params*)

Returns value of mean function derivative wrt the parameters

Method to compute the value of the mean function derivative with respect to the parameters for the inputs and parameters provided. Shapes of `x` and `params` must be consistent based on the return value of the `get_n_params` method. For `FixedMean` classes, there are no parameters so the `params` argument should be an array of length zero. Returns a numpy array of shape `(n_params, x.shape[0])` holding the value of the mean function derivative with respect to each parameter (first axis) for each input point (second axis). Since fixed means have no parameters, this will just be an array of zeros.

Parameters

- **x** (*ndarray*) – Inputs, must be a 1D or 2D numpy array (if 1D a second dimension will be added)
- **params** (*ndarray*) – Parameters, must be a 1D numpy array (of more than 1D will be flattened) and have the same length as the number of parameters required for the provided input

Returns Value of mean function derivative with respect to the parameters evaluated at all input points, numpy array of shape `(n_params, x.shape[0])`

Return type ndarray

mean_f (*x*, *params*)

Returns value of mean function

Method to compute the value of the mean function for the inputs and parameters provided. Shapes of `x` and `params` must be consistent based on the return value of the `get_n_params` method. For `FixedMean` classes, there are no parameters so the `params` argument should be an array of length zero. Returns a numpy array of shape `(x.shape[0],)` holding the value of the mean function for each input point.

Parameters

- **x** (*ndarray*) – Inputs, must be a 1D or 2D numpy array (if 1D a second dimension will be added)
- **params** (*ndarray*) – Parameters, must be a 1D numpy array (of more than 1D will be flattened) and have the same length as the number of parameters required for the provided input (zero in this case)

Returns Value of mean function evaluated at all input points, numpy array of shape `(x.shape[0],)`

Return type ndarray

mean_hessian (*x*, *params*)

Returns value of mean function Hessian wrt the parameters

Method to compute the value of the mean function Hessian with respect to the parameters for the inputs and parameters provided. Shapes of `x` and `params` must be consistent based on the return value of the `get_n_params` method. For `FixedMean` classes, there are no parameters so the `params` argument should be an array of length zero. Returns a numpy array of shape `(n_params, n_params, x.shape[0])` holding the value of the mean function second derivaties with respect to each parameter pair (first twp axes) for each input point (last axis). Since fixed means have no parameters, this will just be an array of zeros.

Parameters

- **x** (*ndarray*) – Inputs, must be a 1D or 2D numpy array (if 1D a second dimension will be added)
- **params** (*ndarray*) – Parameters, must be a 1D numpy array (of more than 1D will be flattened) and have the same length as the number of parameters required for the provided input

Returns Value of mean function Hessian with respect to the parameters evaluated at all input points, numpy array of shape `(n_parmas, n_params, x.shape[0])`

Return type ndarray

mean_inputderiv(*x*, *params*)

Returns value of mean function derivative wrt the inputs

Method to compute the value of the mean function derivative with respect to the inputs for the inputs and parameters provided. Shapes of `x` and `params` must be consistent based on the return value of the `get_n_params` method. For `FixedMean` classes, there are no parameters so the `params` argument should be an array of length zero. Returns a numpy array of shape `(x.shape[1], x.shape[0])` holding the value of the mean function derivative with respect to each input (first axis) for each input point (second axis).

Parameters

- **x** (*ndarray*) – Inputs, must be a 1D or 2D numpy array (if 1D a second dimension will be added)
- **params** (*ndarray*) – Parameters, must be a 1D numpy array (of more than 1D will be flattened) and have the same length as the number of parameters required for the provided input

Returns Value of mean function derivative with respect to the inputs evaluated at all input points, numpy array of shape `(x.shape[1], x.shape[0])`

Return type ndarray

class mogp_emulator.MeanFunction.**ConstantMean**(*val*)

Class representing a constant fixed mean function

Subclass of `FixedMean` where the function is a constant, with the value provided when `ConstantMean` is initialized. Uses utility functions to bind the value to the `fixed_f` function and sets that as the `f` attribute.

Variables

- **f** – fixed mean function, must be callable and take a single argument (the inputs)
- **deriv** – fixed derivative function (optional if no derivatives are needed), must be callable and take a single argument (the inputs)

class mogp_emulator.MeanFunction.**LinearMean**(*index=0*)

Class representing a linear fixed mean function

Subclass of `FixedMean` where the function is a linear function. By default the function is linear in the first input dimension, though any non-negative integer index can be provided to control which input is used in the linear function. Uses utility functions to bind the correct function to the `fixed_f` function and sets that as the `f` attribute and similar with the `fixed_deriv` utility function and the `deriv` attribute.

Variables

- **f** – fixed mean function, must be callable and take a single argument (the inputs)
- **deriv** – fixed derivative function, must be callable and take a single argument (the inputs)

class `mogp_emulator.MeanFunction.`**`Coefficient`**

Class representing a single fitting parameter in a mean function

Class representing a mean function with single free fitting parameter. Does not require any internal state as the parameter value is stored/set externally through fitting routines.

`get_n_params`(*x*)

Determine the number of parameters

Returns the number of parameters for the mean function, which possibly depends on x. For a `Coefficient` class, this is always 1.

Parameters **x** (*ndarray*) – Input array

Returns number of parameters

Return type int

`mean_deriv`(*x*, *params*)

Returns value of mean function derivative wrt the parameters

Method to compute the value of the mean function derivative with respect to the parameters for the inputs and parameters provided. Shapes of `x` and `params` must be consistent based on the return value of the `get_n_params` method. For `Coefficient` classes, the inputs are ignored and the derivative function returns one, broadcasting it appropriately given the shape of the inputs. Returns a numpy array of ones with shape `(1, x.shape[0])` holding the value of the mean function derivative with respect to each parameter (first axis) for each input point (second axis). Since coefficients are single parameters, this will just be an array of ones.

Parameters

- **x** (*ndarray*) – Inputs, must be a 1D or 2D numpy array (if 1D a second dimension will be added)
- **params** (*ndarray*) – Parameters, must be a 1D numpy array (of more than 1D will be flattened) and have the same length as the number of parameters required for the provided input

Returns Value of mean function derivative with respect to the parameters evaluated at all input points, numpy array of shape `(n_params, x.shape[0])`

Return type ndarray

`mean_f`(*x*, *params*)

Returns value of mean function

Method to compute the value of the mean function for the inputs and parameters provided. Shapes of `x` and `params` must be consistent based on the return value of the `get_n_params` method. For `Coefficient` classes, the inputs are ignored and the function returns the value of the parameter broadcasting it appropriately given the shape of the inputs. Thus, the `params` argument should always be an array of length one. Returns a numpy array of shape `(x.shape[0],)` holding the value of the parameter for each input point.

Parameters

- **x** (*ndarray*) – Inputs, must be a 1D or 2D numpy array (if 1D a second dimension will be added)
- **params** (*ndarray*) – Parameters, must be a 1D numpy array (of more than 1D will be flattened) and have the same length as the number of parameters required for the provided input (one in this case)

Returns Value of mean function evaluated at all input points, numpy array of shape `(x.shape[0],)`

Return type ndarray

mean_hessian(*x*, *params*)

Returns value of mean function Hessian wrt the parameters

Method to compute the value of the mean function Hessian with respect to the parameters for the inputs and parameters provided. Shapes of `x` and `params` must be consistent based on the return value of the `get_n_params` method. For `Coefficient` classes, there is only a single parameter so the `params` argument should be an array of length one. Returns a numpy array of shape `(n_params, n_params, x.shape[0])` holding the value of the mean function second derivaties with respect to each parameter pair (first twp axes) for each input point (last axis). Since coefficients depend linearly on a single parameter, this will always be an array of zeros.

Parameters

- **x** (*ndarray*) – Inputs, must be a 1D or 2D numpy array (if 1D a second dimension will be added)
- **params** (*ndarray*) – Parameters, must be a 1D numpy array (of more than 1D will be flattened) and have the same length as the number of parameters required for the provided input

Returns Value of mean function Hessian with respect to the parameters evaluated at all input points, numpy array of shape `(n_parmas, n_params, x.shape[0])`

Return type ndarray

mean_inputderiv(*x*, *params*)

Returns value of mean function derivative wrt the inputs

Method to compute the value of the mean function derivative with respect to the inputs for the inputs and parameters provided. Shapes of `x` and `params` must be consistent based on the return value of the `get_n_params` method. For `Coefficient` classes, there is a single parameters so the `params` argument should be an array of length one. Returns a numpy array of shape `(x.shape[1], x.shape[0])` holding the value of the mean function derivative with respect to each input (first axis) for each input point (second axis). Since coefficients do not depend on the inputs, this is just an array of zeros.

Parameters

- **x** (*ndarray*) – Inputs, must be a 1D or 2D numpy array (if 1D a second dimension will be added)
- **params** (*ndarray*) – Parameters, must be a 1D numpy array (of more than 1D will be flattened) and have the same length as the number of parameters required for the provided input

Returns Value of mean function derivative with respect to the inputs evaluated at all input points, numpy array of shape `(x.shape[1], x.shape[0])`

Return type ndarray

class mogp_emulator.MeanFunction.**PolynomialMean**(*degree*)

Polynomial mean function class

A `PolynomialMean` is a mean function where every input dimension is fit to a fixed degree polynomial. The degree must be provided when creating the class instance. The number of parameters depends on the degree and the shape of the inputs, since a separate set of parameters are used for each input dimension.

Variables **degree** – Polynomial degree, must be a positive integer

get_n_params(*x*)

Determine the number of parameters

Returns the number of parameters for the mean function, which depends on x.

Parameters **x** (*ndarray*) – Input array

Returns number of parameters

Return type int

mean_deriv(*x*, *params*)

Returns value of mean function derivative wrt the parameters

Method to compute the value of the mean function derivative with respect to the parameters for the inputs and parameters provided. Shapes of `x` and `params` must be consistent based on the return value of the `get_n_params` method. Returns a numpy array of shape `(n_params, x.shape[0])` holding the value of the mean function derivative with respect to each parameter (first axis) for each input point (second axis).

Parameters

- **x** (*ndarray*) – Inputs, must be a 1D or 2D numpy array (if 1D a second dimension will be added)
- **params** (*ndarray*) – Parameters, must be a 1D numpy array (of more than 1D will be flattened) and have the same length as the number of parameters required for the provided input

Returns Value of mean function derivative with respect to the parameters evaluated at all input points, numpy array of shape `(n_params, x.shape[0])`

Return type ndarray

mean_f(*x*, *params*)

Returns value of mean function

Method to compute the value of the mean function for the inputs and parameters provided. Shapes of `x` and `params` must be consistent based on the return value of the `get_n_params` method. Returns a numpy array of shape `(x.shape[0],)` holding the value of the mean function for each input point.

Parameters

- **x** (*ndarray*) – Inputs, must be a 1D or 2D numpy array (if 1D a second dimension will be added)
- **params** (*ndarray*) – Parameters, must be a 1D numpy array (of more than 1D will be flattened) and have the same length as the number of parameters required for the provided input

Returns Value of mean function evaluated at all input points, numpy array of shape `(x.shape[0],)`

Return type ndarray

mean_hessian(*x*, *params*)

Returns value of mean function Hessian wrt the parameters

Method to compute the value of the mean function Hessian with respect to the parameters for the inputs and parameters provided. Shapes of `x` and `params` must be consistent based on the return value of the `get_n_params` method. Returns a numpy array of shape `(n_params, n_params, x.shape[0])` holding the value of the mean function second derivaties with respect to each parameter pair (first twp axes) for each input point (last axis). Since polynomial means depend linearly on all input parameters, this will always be an array of zeros.

Parameters

- **x** (*ndarray*) – Inputs, must be a 1D or 2D numpy array (if 1D a second dimension will be added)
- **params** (*ndarray*) – Parameters, must be a 1D numpy array (of more than 1D will be flattened) and have the same length as the number of parameters required for the provided input

Returns Value of mean function Hessian with respect to the parameters evaluated at all input points, numpy array of shape `(n_parmas, n_params, x.shape[0])`

Return type ndarray

mean_inputderiv(*x*, *params*)

Returns value of mean function derivative wrt the inputs

Method to compute the value of the mean function derivative with respect to the inputs for the inputs and parameters provided. Shapes of `x` and `params` must be consistent based on the return value of the `get_n_params` method. Returns a numpy array of shape `(x.shape[1], x.shape[0])` holding the value of the mean function derivative with respect to each input (first axis) for each input point (second axis).

Parameters

- **x** (*ndarray*) – Inputs, must be a 1D or 2D numpy array (if 1D a second dimension will be added)
- **params** (*ndarray*) – Parameters, must be a 1D numpy array (of more than 1D will be flattened) and have the same length as the number of parameters required for the provided input

Returns Value of mean function derivative with respect to the inputs evaluated at all input points, numpy array of shape `(x.shape[1], x.shape[0])`

Return type ndarray

9.6 The `formula` Module

`mogp_emulator.formula.`**mean_from_patsy_formula**(*formula*, *inputdict={}*)

Create a mean function from a patsy formula

This is the functional interface to creating a mean function from a patsy/R formula. The higher level function `MeanFunction` in the `MeanFunction` module is preferred over this one, but this potentially gives the user slightly more control as a patsy `ModelDesc` object can be passed directly, rather than giving a string.

This method takes a string or a patsy `ModelDesc` object as an input and an optional dictionary that map strings to integer indices in the input data. The formula is then parsed with patsy, and the individual terms resulting from that are converted to mean functions and composed using the provided operations.

The string formulas can be specified in several ways. The formula LHS is implicitly always `"y = "` or `"y ~ "`, though these can be explicitly provided as well (though it is ignored in the conversion). The RHS may contain a set of terms containing the add, multiply, power, and call operations much in the same way that the operations would be entered as regular python code. Parentheses are used to indicated prececence as well as the call operation, and square brackets indicate an indexing operation on the inputs. Inputs may be specified as either a string such as `"x[0]"`, `"inputs[0]"`, or a string that can be mapped to an integer index with the optional dictionary passed to the function. Any strings not representing operations or inputs as described above are interpreted as follows: if the string can be converted into a number, then it is interpreted as a `ConstantMean` fixed mean function object; otherwise it is assumed to represent a fitting coefficient. Note that this means many characters that do not represent operations within this mean function language but would not normally be considered as python variables will nonetheless be converted into fitting coefficients – it is up to the user to get this right.

Expressions that are repeated or redundant will not be simplified beyond the parsing done by patsy, so the user should take care that the provided expression is sensible as a mean function and will not cause problems when fitting.

Examples:

```
>>> from mogp_emulator.formula import mean_from_patsy_formula
>>> mf1 = mean_from_patsy_formula("x[0]")
>>> print(mf1)
c + c*x[0]
>>> mf2 = mean_from_patsy_formula("a*b", {"a": 0, "b": 1})
>>> print(mf2)
c + c*x[0] + c*x[1] + c*x[0]*x[1]
```

Parameters

- **formula** (*str or ModelDesc*) – string representing the desired mean function formula or a patsy `ModelDesc` object
- **inputdict** (*dict*) – dictionary used to map variables to input indices. Maps strings to integer indices (must be non-negative). Optional, default is `{}`.

Returns New subclass of `MeanBase` implementing the given formula

Return type subclass of MeanBase (exact type will depend on the formula that is provided)

`mogp_emulator.formula.`**`mean_from_string`**(*formula*, *inputdict={}*)

Create a mean function from a string formula

This is the functional interface to creating a mean function from a string formula. The higher level function `MeanFunction` in the `MeanFunction` module is preferred over this one, but this can also be used and the output is identical.

This method takes a string as an input and an optional dictionary that map strings to integer indices in the input data.

The string formulas can be specified in several ways. The formula LHS is implicitly always `"y = "` or `"y ~ "`, though these can be explicitly provided as well. The RHS may contain a set of terms containing the add, multiply, power, and call operations much in the same way that the operations would be entered as regular python code. Parentheses are used to indicated prececence as well as the call operation, and square brackets indicate an indexing operation on the inputs. Inputs may be specified as either a string such as `"x[0]"`, `"inputs[0]"`, or a string that can be mapped to an integer index with the optional dictionary passed to the function. Any strings not representing operations or inputs as described above are interpreted as follows: if the string can be converted into a number, then it is interpreted as a `ConstantMean` fixed mean function object; otherwise it is assumed to represent a fitting coefficient. Note that this means many characters that do not represent operations within this mean function language but would not normally be considered as python variables will nonetheless be converted into fitting coefficients – it is up to the user to get this right.

Expressions that are repeated or redundant will not be simplified, so the user should take care that the provided expression is sensible as a mean function and will not cause problems when fitting.

Examples:

```
>>> from mogp_emulator.formula import mean_from_string
>>> mf1 = mean_from_string("y = a + b*x[0]")
>>> print(mf1)
```

(continues on next page)

(continued from previous page)

```
c + c*x[0]
>>> mf2 = mean_from_string("c*a*b", {"a": 0, "b": 1})
>>> print(mf2)
c*x[0]*x[1]
```

Parameters

- **formula** (*str*) – string representing the desired mean function formula
- **inputdict** (*dict*) – dictionary used to map variables to input indices. Maps strings to integer indices (must be non-negative). Optional, default is `{}`.

Returns New subclass of `MeanBase` implementing the given formula

Return type subclass of MeanBase (exact type will depend on the formula that is provided)

9.7 The `Kernel` Module

Kernel module, implements a few standard stationary kernels for use with the `GaussianProcess` class. At present, kernels can only be selected manually by setting the `kernel` attribute of the GP. The default is to use the `SquaredExponential` kernel, but this can be changed once the `GaussianProcess` instance is created.

class mogp_emulator.Kernel.**Kernel**

Generic class representing a stationary kernel

This base class implements the necessary scaffolding for defining a stationary kernel. Stationary kernels are only dependent on a distance measure between any two points, so the base class holds all the necessary information for doing the distance computation. Individual subclasses will implement the functional dependence of the kernel on the distance, plus first and second derivatives (if desired) to compute the gradient or Hessian of the kernel with respect to the hyperparameters.

This implementation uses a scaled euclidean distance metric. Each individual parameter has a hyperparameter scale associated with it that is used in the distance computation. If a different metric is to be defined, a new base class needs to be defined that implements the `calc_r`, and optionally `calc_drdtheta` and `calc_d2rdtheta2` methods if gradient or Hessian computation is desired. The methods `kernel_f`, `kernel_gradient`, and `kernel_hessian` can then be used to compute the appropriate quantities with no further modification.

Note that the Kernel object just collates all of the methods together; the class itself does not hold any information on the data point or hyperparamters, which are passed directly to the appropriate methods. Thus, no information needs to be provided when creating a new `Kernal` instance.

calc_K(*r*)

Calculate kernel as a function of distance

This method implements the kernel function as a function of distance. Given an array of distances, this function evaluates the kernel function of those values, returning an array of the same shape. Note that this is not implemented for the base class, as this must be defined for a specific kernel.

Parameters **r** (*array-like*) – Array holding distances between all points. All values in this array must be non-negative.

Returns Array holding kernel evaluations, with the same shape as the input `r`

Return type ndarray

calc_d2Kdr2(*r*)

Calculate second derivative of kernel as a function of distance

This method implements the second derivative of the kernel function as a function of distance. Given an array of distances, this function evaluates the second derivative function of those values, returning an array of the same shape. Note that this is not implemented for the base class, as this must be defined for a specific kernel.

Parameters **r** (*array-like*) – Array holding distances between all points. All values in this array must be non-negative.

Returns Array holding kernel second derivatives, with the same shape as the input `r`

Return type ndarray

calc_d2rdtheta2(*x1*, *x2*, *params*)

Calculate all second derivatives of the distance between all pairs of points with respect to the hyperparameters

This method computes all second derivatives of the scaled Euclidean distance between all pairs of points in `x1` and `x2` with respect to the hyperparameters. The gradient is held in an array with shape `(D, D, n1, n2)`, where `D` is the length of `params`, `n1` is the length of the first axis of `x1`, and `n2` is the length of the first axis of `x2`. This is used in the computation of the gradient and Hessian of the kernel. The first two indices represents the different derivatives with respect to each hyperparameter.

Parameters

- **x1** (*array-like*) – First input array. Must be a 1-D or 2-D array, with the length of the last dimension matching the last dimension of `x2` and one less than the length of `params`. `x1` may be 1-D if either each point consists of a single parameter (and `params` has length 2) or the array only contains a single point (in which case, the array will be reshaped to `(1, D - 1)`).
- **x2** (*array-like*) – Second input array. The same restrictions that apply to `x1` also apply here.
- **params** (*array-like*) – Hyperparameter array. Must be 1-D with length one greater than the last dimension of `x1` and `x2`.

Returns Array holding the second derivatives of the pair-wise distances between points in arrays `x1` and `x2` with respect to the hyperparameters. Will be an array with shape `(D, D, n1, n2)`, where `D` is the length of `params`, `n1` is the length of the first axis of `x1` and `n2` is the length of the first axis of `x2`. The first two axes indicates the different derivative components (i.e. the second derivative with respect to the first parameter is [0,0,:,:], the mixed partial with respect to the first and second parameters is [0,1,:,:] or [1,0,:,:], etc.)

Return type ndarray

calc_dKdr(*r*)

Calculate first derivative of kernel as a function of distance

This method implements the first derivative of the kernel function as a function of distance. Given an array of distances, this function evaluates the derivative function of those values, returning an array of the same shape. Note that this is not implemented for the base class, as this must be defined for a specific kernel.

Parameters **r** (*array-like*) – Array holding distances between all points. All values in this array must be non-negative.

Returns Array holding kernel derivatives, with the same shape as the input `r`

Return type ndarray

calc_drdtheta(*x1*, *x2*, *params*)

Calculate the first derivative of the distance between all pairs of points with respect to the hyperparameters

This method computes the derivative of the scaled Euclidean distance between all pairs of points in `x1` and `x2` with respect to the hyperparameters. The gradient is held in an array with shape `(D, n1, n2)`, where `D` is the length of `params`, `n1` is the length of the first axis of `x1`, and `n2` is the length of the first axis of `x2`. This is used in the computation of the gradient and Hessian of the kernel. The first index represents the different derivatives with respect to each hyperparameter.

Parameters

- **x1** (*array-like*) – First input array. Must be a 1-D or 2-D array, with the length of the last dimension matching the last dimension of `x2` and one less than the length of `params`. `x1` may be 1-D if either each point consists of a single parameter (and `params` has length 2) or the array only contains a single point (in which case, the array will be reshaped to `(1, D - 1)`).
- **x2** (*array-like*) – Second input array. The same restrictions that apply to `x1` also apply here.
- **params** (*array-like*) – Hyperparameter array. Must be 1-D with length one greater than the last dimension of `x1` and `x2`.

Returns Array holding the derivative of the pair-wise distances between points in arrays `x1` and `x2` with respect to the hyperparameters. Will be an array with shape `(D, n1, n2)`, where `D` is the length of `params`, `n1` is the length of the first axis of `x1` and `n2` is the length of the first axis of `x2`. The first axis indicates the different derivative components (i.e. the derivative with respect to the first parameter is [0,:,:], etc.)

Return type ndarray

calc_drdx(*x1*, *x2*, *params*)

Calculate the first derivative of the distance between all pairs of points with respect to the first set of inputs

This method computes the derivative of the scaled Euclidean distance between all pairs of points in `x1` and `x2` with respect to the first input `x1`. The gradient is held in an array with shape `(D - 1, n1, n2)`, where `D` is the length of `params`, `n1` is the length of the first axis of `x1`, and `n2` is the length of the first axis of `x2`. This is used in the computation of the derivative of the kernel with respect to the inputs. The first index represents the different derivatives with respect to each input dimension.

Parameters

- **x1** (*array-like*) – First input array. Must be a 1-D or 2-D array, with the length of the last dimension matching the last dimension of `x2` and one less than the length of `params`. `x1` may be 1-D if either each point consists of a single parameter (and `params` has length 2) or the array only contains a single point (in which case, the array will be reshaped to `(1, D - 1)`).
- **x2** (*array-like*) – Second input array. The same restrictions that apply to `x1` also apply here.
- **params** (*array-like*) – Hyperparameter array. Must be 1-D with length one greater than the last dimension of `x1` and `x2`.

Returns Array holding the derivative of the pair-wise distances between points in arrays `x1` and `x2` with respect to `x1`. Will be an array with shape `(D, n1, n2)`, where `D` is the length of `params`, `n1` is the length of the first axis of `x1` and `n2` is the length of the first axis of `x2`. The first axis indicates the different derivative components (i.e. the derivative with respect to the first input parameter is [0,:,:], etc.)

Return type ndarray

calc_r(*x1*, *x2*, *params*)

Calculate distance between all pairs of points

This method computes the scaled Euclidean distance between all pairs of points in `x1` and `x2`. Each component distance is multiplied by the corresponding hyperparameter prior to summing and taking the square root. For example, if `x1 = [1.]`, `x2` = [2.], and `params = [2., 2.]` then `calc_r` would return $\sqrt{2(1-2)^2} = \sqrt{2}$ as an array with shape `(1,1)`.

Parameters

- **x1** (*array-like*) – First input array. Must be a 1-D or 2-D array, with the length of the last dimension matching the last dimension of `x2` and one less than the length of `params`. `x1` may be 1-D if either each point consists of a single parameter (and `params` has length 2) or the array only contains a single point (in which case, the array will be reshaped to `(1, D - 1)`).
- **x2** (*array-like*) – Second input array. The same restrictions that apply to `x1` also apply here.
- **params** (*array-like*) – Hyperparameter array. Must be 1-D with length one greater than the last dimension of `x1` and `x2`.

Returns Array holding all pair-wise distances between points in arrays `x1` and `x2`. Will be an array with shape `(n1, n2)`, where `n1` is the length of the first axis of `x1` and `n2` is the length of the first axis of `x2`.

Return type ndarray

kernel_deriv(*x1*, *x2*, *params*)

Compute kernel gradient for a set of inputs

Returns the value of the kernel gradient for two sets of input points and a choice of hyperparameters. This function should not need to be modified for different choices of the kernel function or distance metric, as after checking the inputs it simply calls the routine to compute the distance metric, kernel function, and the appropriate derivative functions of the distance and kernel functions.

Parameters

- **x1** (*array-like*) – First input array. Must be a 1-D or 2-D array, with the length of the last dimension matching the last dimension of `x2` and one less than the length of `params`. `x1` may be 1-D if either each point consists of a single parameter (and `params` has length 2) or the array only contains a single point (in which case, the array will be reshaped to `(1, D - 1)`).
- **x2** (*array-like*) – Second input array. The same restrictions that apply to `x1` also apply here.
- **params** (*array-like*) – Hyperparameter array. Must be 1-D with length one greater than the last dimension of `x1` and `x2`.

Returns Array holding the gradient of the kernel function between points in arrays `x1` and `x2` with respect to the hyperparameters. Will be an array with shape `(D, n1, n2)`, where `D` is the length of `params`, `n1` is the length of the first axis of `x1` and `n2` is the length of the first axis of `x2`. The first axis indicates the different derivative components (i.e. the derivative with respect to the first parameter is [0,:,:], etc.)

Return type ndarray

kernel_f(*x1*, *x2*, *params*)

Compute kernel values for a set of inputs

Returns the value of the kernel for two sets of input points and a choice of hyperparameters. This function should not need to be modified for different choices of the kernel function or distance metric, as after checking the inputs it simply calls the routine to compute the distance metric and then evaluates the kernel function for those distances.

Parameters

- **x1** (*array-like*) – First input array. Must be a 1-D or 2-D array, with the length of the last dimension matching the last dimension of `x2` and one less than the length of `params`. `x1` may be 1-D if either each point consists of a single parameter (and `params` has length 2) or the array only contains a single point (in which case, the array will be reshaped to `(1, D - 1)`).
- **x2** (*array-like*) – Second input array. The same restrictions that apply to `x1` also apply here.
- **params** (*array-like*) – Hyperparameter array. Must be 1-D with length one greater than the last dimension of `x1` and `x2`.

Returns Array holding all kernel values between points in arrays `x1` and `x2`. Will be an array with shape `(n1, n2)`, where `n1` is the length of the first axis of `x1` and `n2` is the length of the first axis of `x2`.

Return type ndarray

kernel_hessian (*x1*, *x2*, *params*)

Calculate the Hessian of the kernel evaluated for all pairs of points with respect to the hyperparameters

Returns the value of the kernel Hessian for two sets of input points and a choice of hyperparameters. This function should not need to be modified for different choices of the kernel function or distance metric, as after checking the inputs it simply calls the routine to compute the distance metric, kernel function, and the appropriate derivative functions of the distance and kernel functions.

Parameters

- **x1** (*array-like*) – First input array. Must be a 1-D or 2-D array, with the length of the last dimension matching the last dimension of `x2` and one less than the length of `params`. `x1` may be 1-D if either each point consists of a single parameter (and `params` has length 2) or the array only contains a single point (in which case, the array will be reshaped to `(1, D - 1)`).
- **x2** (*array-like*) – Second input array. The same restrictions that apply to `x1` also apply here.
- **params** (*array-like*) – Hyperparameter array. Must be 1-D with length one greater than the last dimension of `x1` and `x2`.

Returns Array holding the Hessian of the pair-wise distances between points in arrays `x1` and `x2` with respect to the hyperparameters. Will be an array with shape `(D, D, n1, n2)`, where `D` is the length of `params`, `n1` is the length of the first axis of `x1` and `n2` is the length of the first axis of `x2`. The first two axes indicates the different derivative components (i.e. the second derivative with respect to the first parameter is [0,0,:,:], the mixed partial with respect to the first and second parameters is [0,1,:,:] or [1,0,:,:], etc.)

Return type ndarray

kernel_inputderiv (*x1*, *x2*, *params*)

Compute derivative of Kernel with respect to inputs x1

Returns the value of the kernel derivative with respect to the first set of input points given inputs and a choice of hyperparameters. This function should not need to be modified for different choices of the kernel function or distance metric, as after checking the inputs it simply calls the routine to compute the distance metric, kernel function, and the appropriate derivative functions of the distance and kernel functions.

Parameters

- **x1** (*array-like*) – First input array. Must be a 1-D or 2-D array, with the length of the last dimension matching the last dimension of `x2` and one less than the length of `params`. `x1` may be 1-D if either each point consists of a single parameter (and `params` has length 2) or the array only contains a single point (in which case, the array will be reshaped to `(1, D - 1)`).
- **x2** (*array-like*) – Second input array. The same restrictions that apply to `x1` also apply here.
- **params** (*array-like*) – Hyperparameter array. Must be 1-D with length one greater than the last dimension of `x1` and `x2`.

Returns Array holding the derivative of the kernel function between points in arrays `x1` and `x2` with respect to the first inputs `x1`. Will be an array with shape `(D, n1, n2)`, where `D` is the length of `params`, `n1` is the length of the first axis of `x1` and `n2` is the length of the first axis of `x2`. The first axis indicates the different derivative components (i.e. the derivative with respect to the first input dimension is [0,:,:], etc.)

Return type ndarray

class `mogp_emulator.Kernel.`**`SquaredExponential`**

Implementation of the squared exponential kernel

Class representing a squared exponential kernel. It derives from the base class for a stationary kernel, using the scaled Euclidean distance metric. The subclass then just defines the kernel function and its derivatives.

`calc_K`(*r*)

Compute K(r) for the squared exponential kernel

This method implements the squared exponential kernel function as a function of distance. Given an array of distances, this function evaluates the kernel function of those values, returning an array of the same shape.

Parameters **r** (*array-like*) – Array holding distances between all points. All values in this array must be non-negative.

Returns Array holding kernel evaluations, with the same shape as the input `r`

Return type ndarray

`calc_d2Kdr2`(*r*)

Calculate second derivative of the squared exponential kernel as a function of distance

This method implements the second derivative of the squared exponential kernel function as a function of distance. Given an array of distances, this function evaluates the second derivative function of those values, returning an array of the same shape.

Parameters **r** (*array-like*) – Array holding distances between all points. All values in this array must be non-negative.

Returns Array holding kernel second derivatives, with the same shape as the input `r`

Return type ndarray

`calc_dKdr`(*r*)

Calculate first derivative of the squared exponential kernel as a function of distance

This method implements the first derivative of the squared exponential kernel function as a function of distance. Given an array of distances, this function evaluates the derivative function of those values, returning an array of the same shape.

Parameters **r** (*array-like*) – Array holding distances between all points. All values in this array must be non-negative.

Returns Array holding kernel derivatives, with the same shape as the input `r`

Return type ndarray

class mogp_emulator.Kernel.**Matern52**

Implementation of the Matern 5/2 kernel

Class representing the Matern 5/2 kernel. It derives from the base class for a stationary kernel, using the scaled Euclidean distance metric. The subclass then just defines the kernel function and its derivatives.

calc_K(*r*)

Compute K(r) for the Matern 5/2 kernel

This method implements the Matern 5/2 kernel function as a function of distance. Given an array of distances, this function evaluates the kernel function of those values, returning an array of the same shape.

Parameters **r** (*array-like*) – Array holding distances between all points. All values in this array must be non-negative.

Returns Array holding kernel evaluations, with the same shape as the input `r`

Return type ndarray

calc_d2Kdr2(*r*)

Calculate second derivative of the squared exponential kernel as a function of distance

This method implements the second derivative of the squared exponential kernel function as a function of distance. Given an array of distances, this function evaluates the second derivative function of those values, returning an array of the same shape.

Parameters **r** (*array-like*) – Array holding distances between all points. All values in this array must be non-negative.

Returns Array holding kernel second derivatives, with the same shape as the input `r`

Return type ndarray

calc_dKdr(*r*)

Calculate first derivative of the Matern 5/2 kernel as a function of distance

This method implements the first derivative of the Matern 5/2 kernel function as a function of distance. Given an array of distances, this function evaluates the derivative function of those values, returning an array of the same shape.

Parameters **r** (*array-like*) – Array holding distances between all points. All values in this array must be non-negative.

Returns Array holding kernel derivatives, with the same shape as the input `r`

Return type ndarray

9.8 The `Priors` Module

class mogp_emulator.Priors.**Prior**

Generic Prior Object

d2logpdtheta2(*x*)
Computes second derivative of log probability at a given value

dlogpdtheta(*x*)
Computes derivative of log probability at a given value

logp(*x*)
Computes log probability at a given value

class mogp_emulator.Priors.**NormalPrior**(*mean*, *std*)
Normal Distribution Prior object

Admits input values from -inf/+inf, and no transformations are assumed. Thus, for mean function hyperparameters this produces a normal distribution with given mean and variance, and for covariance/nugget hyperparameters this produces a lognormal distribution with given log mean and variance.

Take two parameters: mean and std. mean can take any numeric value, while std must be positive.

d2logpdtheta2(*x*)
Computes second derivative of log probability at a given value

dlogpdtheta(*x*)
Computes derivative of log probability at a given value

logp(*x*)
Computes log probability at a given value

class mogp_emulator.Priors.**GammaPrior**(*shape*, *scale*)
Gamma Distribution Prior object

Admits input values from -inf/+inf assumed on a logarithmic scale, and transforms by taking the exponential of the input. Thus, this is assumed to be appropriate for covariance hyperparameters where such transformations are assumed when computing the covariance function.

Take two parameters: shape α and scale β. Both must be positive, and they are defined such that

$p(x) = \frac{\beta^{-\alpha} x^{\beta-1}}{\Gamma(/alpha)} \exp(-x/\beta)$

d2logpdtheta2(*x*)
Computes second derivative of log probability at a given value

dlogpdtheta(*x*)
Computes derivative of log probability at a given value

logp(*x*)
Computes log probability at a given value

class mogp_emulator.Priors.**InvGammaPrior**(*shape*, *scale*)
Inverse Gamma Distribution Prior object

Admits input values from -inf/+inf assumed on a logarithmic scale, and transforms by taking the exponential of the input. Thus, this is assumed to be appropriate for covariance hyperparameters where such transformations are assumed when computing the covariance function.

Take two parameters: shape α and scale β. Both must be positive, and they are defined such that

$p(x) = \frac{\beta^{\alpha} x^{-\beta-1}}{\Gamma(/alpha)} \exp(-\beta/x)$

d2logpdtheta2(*x*)
Computes second derivative of log probability at a given value

dlogpdtheta(*x*)
Computes derivative of log probability at a given value

logp(*x*)
Computes log probability at a given value

9.9 The `DimensionReduction` module

This module provides classes and utilities for performing dimension reduction. Currently there is a single class *mogp_emulator.gKDR* which implements the method of Fukumizu and Leng [FL13].

Example:

```
>>> from mogp_emulator import gKDR
>>> import numpy as np
>>> X = np.array([[0.0, 0.0], [0.0, 1.0], [1.0, 0.0], [1.0, 1.0]])
>>> Y = np.array([0.0, 1.0, 5.0, 6.0])
>>> xnew = np.array([0.5, 0.5])
>>> dr = gKDR(X, Y, 1)
>>> dr(xnew)
array([0.60092477])
```

In this example, the reduction was performed from a two- to a one-dimensional input space. The value returned by `dr(xnew)` is the input coordinate *xnew* transformed to the reduced space.

The following example illustrates how to perform Gaussian process regression on the reduced input space:

```
>>> import numpy as np
>>> from mogp_emulator import gKDR, GaussianProcess, fit_GP_MAP

### generate some training data (from the function f)

>>> def f(x):
...     return np.sqrt(x[0] + np.sin(0.1 * x[1]))

>>> X = np.mgrid[0:10,0:10].T.reshape(-1,2)/10.0
>>> print(X)
[[0.  0. ]
 [0.1 0. ]
 [0.2 0. ]
 [0.3 0. ]
 [0.4 0. ]
 [0.5 0. ]
 [0.6 0. ]
 [0.7 0. ]
 [0.8 0. ]
 [0.9 0. ]
 [0.  0.1]
 [0.1 0.1]
 ...
 [0.8 0.9]
 [0.9 0.9]]

>>> Y = np.apply_along_axis(f, 1, X)

### reduced input space
>>> dr = gKDR(X, Y, K=1)

### train a Gaussian Process with reduced inputs
```

(continues on next page)

(continued from previous page)

```
>>> gp = GaussianProcess(dr(X), Y)
>>> gp = fit_GP_MAP(gp)

### make a prediction (given an input in the reduced space)
>>> Xnew = np.array([0.12, 0.37])
>>> gp.predict(dr(Xnew))
(array([0.398083]), ...)

>>> f(Xnew)
0.396221
```

9.9.1 Dimension Reduction Classes

class mogp_emulator.**gKDR**(*X*, *Y*, *K=None*, *X_scale=1.0*, *Y_scale=1.0*, *EPS=1e-08*, *SGX=None*, *SGY=None*)

Dimension reduction by the gKDR method.

See link [Fukumizu1] (and in particular, [FL13]) for details of the method.

Note that this is a simpler and faster method than the original "KDR" method by the same authors (but with an added approximation). The KDR method will be implemented separately.

An instance of this class is callable, with the `__call__` method taking an input coordinate and mapping it to a reduced coordinate.

Note that this class currently implements a *direct* translation of the Matlab implementation of KernelDeriv (see link above) into Python/NumPy. It is due to be replaced with a Fortran implementation, but this should not affect the interface.

__init__(*X*, *Y*, *K=None*, *X_scale=1.0*, *Y_scale=1.0*, *EPS=1e-08*, *SGX=None*, *SGY=None*)

Create a gKDR object

Given some M-dimensional inputs (explanatory variables) X, and corresponding one-dimensional outputs (responses) Y, use the gKDR method to produce a reduced version of the input space with K dimensions.

Parameters

- **X** (*ndarray, of shape (N, M)*) – N rows of M dimensional input vectors
- **Y** (*ndarray, of shape (N,)*) – N response values
- **K** (*integer*) – The number of reduced dimensions to use ($0 <= K <= M$).
- **EPS** (*float*) – The regularization parameter, default *1e-08*; *EPS >= 0*
- **X_scale** (*float*) – Optional, default *1.0*. If SGX is None (the default), scale the automatically determined value for SGX by X_scale. Otherwise ignored.
- **Y_scale** (*float*) – Optional, default *1.0*. If SGY is None (the default), scale the automatically determined value for SGY by Y_scale. Otherwise ignored.
- **SGX** (*float | NoneType*) – Optional, default *None*. The kernel parameter representing the scale of variation on the input space. If *None*, then the median distance between pairs of input points (X) is used (as computed by `mogp_emulator.DimensionReduction.median_dist()`). If a float is passed, then this must be positive.
- **SGY** (*float | NoneType*) – Optional, default *None*. The kernel parameter representing the scale of variation on the output space. If *None*, then the median distance between pairs of output values (Y) is used (as computed by `mogp_emulator.`

DimensionReduction.median_dist()). If a float is passed, then this must be positive.

`__call__`(*X*)

Calling a gKDR object with a vector of N inputs returns the inputs mapped to the reduced space.

Parameters **`X`** (ndarray, of shape *(N, M)*) – *N* coordinates (rows) in the unreduced *M*-dimensional space

Return type ndarray, of shape *(N, K)*

Returns *N* coordinates (rows) in the reduced *K*-dimensional space

`classmethod tune_parameters`(*X*, *Y*, *train_model*, *cXs=None*, *cYs=None*, *maxK=None*, *cross_validation_folds=5*, *verbose=False*)

Constructs a gKDR model with the structural dimension (K) and kernel scale parameters (cX, cY) that approximately minimize the L1 error between Y and the trained model (resulting from calling *train_model* on X and Y).

Currently, this works as follows. For each choice of cX and cY in *cXs* and *cYs*, find K by starting from a K of 1 and doubling K until the loss increases (using the value of K just before), or until K equals the input dimension (or maxK if specified). The resulting choice of *(cX, cY, K)* is then taken as the minimum such choice over the cX, cY.

Parameters

- **`X`** (*`ndarray, of shape (N, M)`*) – *N* input points with dimension *M*
- **`Y`** (*`ndarray, of shape (N,)`*) – the *N* model observations, corresponding to each
- **`train_model`** (callable with the signature *(ndarray, ndarray) -> ndarray -> ndarray*) – a callable, that when called with model inputs X (shape *(Ntrain, M)*) and Y (shape *(Ntrain, M)*), returns a "model", which is another callable, taking an array (shape *(Npredict, M)*) of the points where a prediction is desired, and returning an array (shape *(Npredict,)*) of the corresponding predictions.
- **`cXs`** (Iterable of *float*, or *NoneType*) – (optional, default None). The scale parameter for *X* in the dimension reduction kernel. Passed as the parameter *X_scale* to the gKDR constructor (*mogp_emulator.gKDR.__init__()*). If None, *[0.5, 1, 5.0]* is used.
- **`cYs`** (Iterable of *float*, or *NoneType*) – (optional, default None). The scale parameter for *Y* in the dimension reduction kernel. Passed as the parameter *Y_scale* to the gKDR constructor (*mogp_emulator.gKDR.__init__()*). If None, *[0.5, 1, 5.0]* is used.
- **`maxK`** (*`integer, or NoneType`*) – (optional default *None*). The largest structural dimension to consider in the optimization. This is useful when there is a known bound on the dimension, to stop e.g. poor values of cX or cY needlessly extending the search. It is a good idea to choose this parameter generously.
- **`cross_validation_folds`** (*`integer`*) – (optional, default is 5): Use this many folds for cross-validation when tuning the parameters.
- **`verbose`** (*`bool`*) – produce a log to stdout of the optimization?

Returns A pair of: the gKDR object with parameters tuned according to the above method, and a number representing the L1 loss of the model trained on inputs as reduced by this dimension reduction object. :rtype: pair of a gKDR and a non-negative float

Example

Tune the structural dimension and lengthscale parameters within the kernel, minimizing the the loss from a Gaussian process regression:

```
>>> from mogp_emulator import gKDR
>>> from mogp_emulator import GaussianProcess
>>> X = ...
>>> Y = ...
>>> dr, loss = gKDR.tune_parameters(X, Y, GaussianProcess.train_model)
>>> gp = GaussianProcess(dr(X), Y)
```

Or, specifying some optional parameters for the lengthscales, the maximum value of *K* to use, the number of folds for cross-validation, and producing verbose output:

```
>>> dr, loss = gKDR.tune_parameters(X, Y, GaussianProcess.train_model,
...                                 cXs = [0.5, 1.0, 2.0], cYs = [2.0],
...                                 maxK = 25, cross_validation_folds=4,␣
↪verbose = True)
```

9.9.2 Utilities

DimensionReduction.**gram_matrix**(*k*)

Computes the Gram matrix of *X*

Parameters

- **X** (*ndarray*) – Two-dimensional numpy array, where rows are feature vectors
- **k** – The covariance function

Returns The gram matrix of *X* under the kernel *k*, that is, $G_{ij} = k(X_i, X_j)$

DimensionReduction.**gram_matrix_sqexp**(*sigma2*)

Computes the Gram matrix of *X* under the squared expontial kernel. Equivalent to, but more efficient than, calling `gram_matrix(X, k_sqexp)`

Parameters

- **X** (*ndarray*) – Two-dimensional numpy array, where rows are feature vectors
- **sigma2** – The variance parameter of the squared exponential kernel

Returns The gram matrix of *X* under the squared exponential kernel *k_sqexp* with variance parameter *sigma2* ($= \sigma^2$), that is, $G_{ij} = k_{sqexp}(X_i, X_j; \sigma^2)$

DimensionReduction.**median_dist**()

Return the median of the pairwise (Euclidean) distances between each row of X

References

9.10 The `ExperimentalDesign` Class

Base class representing a generic one-shot design of experiments with uncorrelated parameters

This class provides the base implementation for a class for designing experiments to sample the parameter space of a complex model. The parameter space can be specified in a variety of ways, but essentially the user must provide a Probability Point Function (PPF, or inverse of the Cumulative Distribution Function) for each input parameter. Each PPF function takes a single numeric input and maps from the interval $[0, 1]$ to the desired parameter distribution value for a given parameter, and each parameter has a separate function describing its distribution. Note that this makes the assumption of no correlations between any of the parameter values (a future version may implement an experimental design where there are such parameter correlations). Once the design is initialized, a desired number of samples can be drawn from the design, returning an array holding the desired number of samples from the parameter space.

Internally, the class holds the set of PPFs for all of the parameter values, and samples are drawn by calling the `sample` method. To draw the samples, a specific method `_draw_samples` must be defined that generates a series of points in the $[0, 1]^n$ hypercube, where n is the number of paramters. This set of samples from the hypercube is then mapped to the parameter space using the given PPF functions. Thus, defining a new design protocol only requires defining a new `_draw_samples` method and redefining the `__init__` method to set the internal `method` attribute. By default, no `_draw_samples` method is defined, so the base `ExperimentalDesign` class is only intended to be used to define new protocols (trying to sample from an `ExperimentalDesign` instance will return a `NotImplementedError`).

class `mogp_emulator.ExperimentalDesign.`**`ExperimentalDesign`**`(`**args*`)`

Base class representing a generic one-shot design of experiments with uncorrelated parameters

This class provides the base implementation for a class for designing experiments to sample the parameter space of a complex model. The parameter space can be specified in a variety of ways, but essentially the user must provide a Probability Point Function (PPF, or inverse of the Cumulative Distribution Function) for each input parameter. Each PPF function takes a single numeric input and maps from the interval $[0, 1]$ to the desired parameter distribution value for a given parameter, and each parameter has a separate function describing its distribution. Note that this makes the assumption of no correlations between any of the parameter values (a future version may implement an experimental design where there are such parameter correlations). Once the design is initialized, a desired number of samples can be drawn from the design, returning an array holding the desired number of samples from the parameter space.

Internally, the class holds the set of PPFs for all of the parameter values, and samples are drawn by calling the `sample` method. To draw the samples, a specific method `_draw_samples` must be defined that generates a series of points in the $[0, 1]^n$ hypercube, where n is the number of paramters. This set of samples from the hypercube is then mapped to the parameter space using the given PPF functions. Thus, defining a new design protocol only requires defining a new `_draw_samples` method and redefining the `__init__` method to set the internal `method` attribute. By default, no `_draw_samples` method is defined, so the base `ExperimentalDesign` class is only intended to be used to define new protocols (trying to sample from an `ExperimentalDesign` instance will return a `NotImplementedError`).

`__init__``(`**args*`)`

Create a new instance of an experimental design

Creates a new instance of a design of experiments, which draws samples from the parameter space of a complex model. It is often used to generate data for a Gaussian Process emulator to fit the outputs of the complex model. This is a base class that does not implement the method for sampling from the distribution; to use an experimental design in practice you should use one of the derived classes provided or create your own.

The experimental design can be initialized in several ways depending on the arguments provided, ranging from the simplest to the most complicated.

1. Provide an integer `n` indicating the number of input parameters. If this is used to create an instance, it is assumed that all parameters are unformly distributed over the n-dimensional hypercube.
2. Provide an integer `n` and a tuple `(a, b)` of length 2 containing two numeric values (where $a < b$). In this case, all parameters are assumed to be uniformly distributed over the interval $[a, b]$.
3. Provide an integer `n` and a function that takes a single numeric input in the interval $[0, 1]$ and maps it to the parameter space. In this case, all parameters are assumed to follow the provided Probability Point Function.
4. Provide a list of tuples of length 2 containing numeric values (as above, the first number must smaller than the second number). The design then assumes that the number of parameters is the length of the

list, and each parameter follows a uniform distribution with the bounds given by the respective tuple in the given list.

5. Provide a list of functions taking a single input (as above, each function must map the interval $[0, 1]$ to the parameter space). The number of parameters in the design is the length of the list, and the given PPF functions define the parameter space for each input.

More concretely, if one input parameter is given, you may initilize the class in any of the following ways:

Parameters **`n_parameters`** (*`int`*) – Integer specifying the number of parameters (must be positive). The design will sample each parameter over the interval $[0, 1]$

or

Parameters **`bounds_list`** (*`list`*) – List of tuples containing two numeric values, each of which has the smaller number first. Each parameter then takes a uniform distribution with bounds given by each tuple.

or

Parameters **`ppf_list`** (*`list`*) – List of functions or other callable, each of which accepts one argument and maps the interval $[0, 1]$ to the parameter space. Each parameter follows the distribution given by the respective PPF function.

and if two input parameters are given:

Parameters

- **`n_parameters`** (*`int`*) – Integer specifying the number of parameters (must be positive). The design will sample each parameter over the interval $[0, 1]$
- **`bounds`** (*`tuple`*) – Tuple or other iterable containing two numeric values, where the smaller number must come first. Each parameter then takes a uniform distribution with bounds given by the numbers provided in the tuple.

or

Parameters

- **`n_parameters`** (*`int`*) – Integer specifying the number of parameters (must be positive). The design will sample each parameter over the interval $[0, 1]$
- **`ppf`** (*`function`*) – Function or other callable, which accepts one argument and maps the interval $[0, 1]$ to the parameter space. Each parameter follows the distribution given by the PPF function.

The `scipy.stats` package provides implementations of a wide range of distributions, with pre-defined PPF functions. See the Scipy user manual for more details. Note that in order to get a compatible PPF function that only takes a single input, you will need to set any parameters needed to define the distribution.

Internally, the class defines any PPF functions based on the input data and collects all of the PPF functions in a list. The class also contains information on the method used to draw samples from the design.

To create a usable implementation based on this class, the user must define the method `_draw_samples`, which takes a positive integer input `n_samples` and draws `n_samples` from the $[0, 1]^n$ hypercube, where n is the number of parameters. The user must also modify the `method` attribute of the design in order to have the `__str__` method work correctly. All other functionality should not require any changes from the base class.

`get_method``()`

Returns the method used to draw samples from the design

This method returns the method used to draw samples from the experimental design. The base class does not implement a method, so if you try to call this on the base class the code will raise a

`NotImplementedError`. When deriving new designs from the base class, the method should be set when calling the `__init__` method.

Returns Method used to draw samples from the design.

Return type str

get_n_parameters()

Returns number of parameters in the experimental design

This method returns the number of parameters in the experimental design. This is set when initializing the object, an cannot be modified.

Returns Number of parameters in the experimental design.

Return type int

sample(*n_samples*)

Draw parameter samples from the experimental design

This method implements drawing parameter samples from the experimental design. The method does this by calling the `_draw_samples` method to obtain samples from the $[0, 1]^n$ hypercube, where n is the number of parameters. The `sample``method then transforms these samples drawn from the low level method to the actual parameter values using the PPF functions provided when initilizing the object. Note that this method also checks that all parameter values are finite; if any ``NaN` values are returned, an error will be raised.

Note that by implementing the sampling in this way, modifications to the method to draw samples using a different protocol only needs to change the `_draw_samples` method. This makes it simpler to define new designs, as only a single method needs to be altered.

Parameters **n_samples** (*int*) – Number of samples to be drawn from the design (must be a positive integer)

Returns Samples drawn from the design parameter space as a numpy array with shape `(n_samples, n_parameters)`

Return type ndarray

9.11 The MonteCarloDesign Class

Class representing a one-shot design of experiments with uncorrelated parameters using Monte Carlo Sampling

This class provides an implementation for a class for designing experiments to sample the parameter space of a complex model using random Monte Carlo sampling. The parameter space can be specified in a variety of ways, but essentially the user must provide a Probability Point Function (PPF, or inverse of the Cumulative Distribution Function) for each input parameter. Each PPF function takes a single numeric input and maps from the interval $[0, 1]$ to the desired parameter distribution value for a given parameter, and each parameter has a separate function describing its distribution. Note that this makes the assumption of no correlations between any of the parameter values (a future version may implement an experimental design where there are such parameter correlations). Once the design is initialized, a desired number of samples can be drawn from the design, returning an array holding the desired number of samples from the parameter space.

Internally, the class holds the set of PPFs for all of the parameter values, and samples are drawn by calling the `sample` method. To draw the samples, the `_draw_samples` is used to generate a series of points in the $[0, 1]^n$ hypercube using random Monte Carlo sampling, where n is the number of paramters. This set of samples from the hypercube is then mapped to the parameter space using the given PPF functions.

This design does not attempt to uniformly sample the space, but rather just makes random draws from the parameter distributions. For this reason, small designs using this method may not sample the parameter space very efficiently. However, generating a large number of samples using this protocol can be done very efficiently, as drawing a sample only requires generating a series of pseduorandom numbers.

class `mogp_emulator.ExperimentalDesign.`**`MonteCarloDesign`**`(`**args*`)`

Class representing a one-shot design of experiments with uncorrelated parameters using Monte Carlo Sampling

This class provides an implementation for a class for designing experiments to sample the parameter space of a complex model using random Monte Carlo sampling. The parameter space can be specified in a variety of ways, but essentially the user must provide a Probability Point Function (PPF, or inverse of the Cumulative Distribution Function) for each input parameter. Each PPF function takes a single numeric input and maps from the interval $[0, 1]$ to the desired parameter distribution value for a given parameter, and each parameter has a separate function describing its distribution. Note that this makes the assumption of no correlations between any of the parameter values (a future version may implement an experimental design where there are such parameter correlations). Once the design is initialized, a desired number of samples can be drawn from the design, returning an array holding the desired number of samples from the parameter space.

Internally, the class holds the set of PPFs for all of the parameter values, and samples are drawn by calling the `sample` method. To draw the samples, the `_draw_samples` is used to generate a series of points in the $[0, 1]^n$ hypercube using random Monte Carlo sampling, where n is the number of paramters. This set of samples from the hypercube is then mapped to the parameter space using the given PPF functions.

This design does not attempt to uniformly sample the space, but rather just makes random draws from the parameter distributions. For this reason, small designs using this method may not sample the parameter space very efficiently. However, generating a large number of samples using this protocol can be done very efficiently, as drawing a sample only requires generating a series of pseduorandom numbers.

`__init__``(`**args*`)`

Create a new instance of a Monte Carlo experimental design

Creates a new instance of a Monte Carlo design of experiments, which draws samples randomly from the parameter space of a complex model. It can be used to generate data for a Gaussian Process emulator to fit the outputs of the complex model. Because the samples are drawn randomly, small designs may not sample the space very well, but the samples can be drawn quickly as they are entirely random.

The experimental design can be initialized in several ways depending on the arguments provided, ranging from the simplest to the most complicated.

1. Provide an integer `n` indicating the number of input parameters. If this is used to create an instance, it is assumed that all parameters are unformly distributed over the n-dimensional hypercube.
2. Provide an integer `n` and a tuple `(a, b)` of length 2 containing two numeric values (where $a < b$). In this case, all parameters are assumed to be uniformly distributed over the interval $[a, b]$.
3. Provide an integer `n` and a function that takes a single numeric input in the interval $[0, 1]$ and maps it to the parameter space. In this case, all parameters are assumed to follow the provided Probability Point Function.
4. Provide a list of tuples of length 2 containing numeric values (as above, the first number must smaller than the second number). The design then assumes that the number of parameters is the length of the list, and each parameter follows a uniform distribution with the bounds given by the respective tuple in the given list.
5. Provide a list of functions taking a single input (as above, each function must map the interval $[0, 1]$ to the parameter space). The number of parameters in the design is the length of the list, and the given PPF functions define the parameter space for each input.

More concretely, if one input parameter is given, you may initilize the class in any of the following ways:

Parameters **n_parameters** (*int*) – Integer specifying the number of parameters (must be positive). The design will sample each parameter over the interval $[0, 1]$

or

Parameters **bounds_list** (*list*) – List of tuples containing two numeric values, each of which has the smaller number first. Each parameter then takes a uniform distribution with bounds given by each tuple.

or

Parameters **ppf_list** (*list*) – List of functions or other callable, each of which accepts one argument and maps the interval $[0, 1]$ to the parameter space. Each parameter follows the distribution given by the respective PPF function.

and if two input parameters are given:

Parameters

- **n_parameters** (*int*) – Integer specifying the number of parameters (must be positive). The design will sample each parameter over the interval $[0, 1]$
- **bounds** (*tuple*) – Tuple or other iterable containing two numeric values, where the smaller number must come first. Each parameter then takes a uniform distribution with bounds given by the numbers provided in the tuple.

or

Parameters

- **n_parameters** (*int*) – Integer specifying the number of parameters (must be positive). The design will sample each parameter over the interval $[0, 1]$
- **ppf** (*function*) – Function or other callable, which accepts one argument and maps the interval $[0, 1]$ to the parameter space. Each parameter follows the distribution given by the PPF function.

The `scipy.stats` package provides implementations of a wide range of distributions, with pre-defined PPF functions. See the Scipy user manual for more details. Note that in order to get a compatible PPF function that only takes a single input, you will need to set any parameters needed to define the distribution.

Internally, the class defines any PPF functions based on the input data and collects all of the PPF functions in a list. The class also contains information on the method used to draw samples from the design.

get_method()

Returns the method used to draw samples from the design

This method returns the method used to draw samples from the experimental design. The base class does not implement a method, so if you try to call this on the base class the code will raise a `NotImplementedError`. When deriving new designs from the base class, the method should be set when calling the `__init__` method.

Returns Method used to draw samples from the design.

Return type str

get_n_parameters()

Returns number of parameters in the experimental design

This method returns the number of parameters in the experimental design. This is set when initializing the object, an cannot be modified.

Returns Number of parameters in the experimental design.

Return type int

sample(*n_samples*)

Draw parameter samples from the experimental design

This method implements drawing parameter samples from the experimental design. The method does this by calling the `_draw_samples` method to obtain samples from the $[0, 1]^n$ hypercube, where n is the number of parameters. The `sample``method then transforms these samples drawn from the low level method to the actual parameter values using the PPF functions provided when initilizing the object. Note that this method also checks that all parameter values are finite; if any ``NaN` values are returned, an error will be raised.

Note that by implementing the sampling in this way, modifications to the method to draw samples using a different protocol only needs to change the `_draw_samples` method. This makes it simpler to define new designs, as only a single method needs to be altered.

Parameters **n_samples** (*int*) – Number of samples to be drawn from the design (must be a positive integer)

Returns Samples drawn from the design parameter space as a numpy array with shape `(n_samples, n_parameters)`

Return type ndarray

9.12 The `LatinHypercubeDesign` Class

Class representing a one-shot design of experiments with uncorrelated parameters using Latin Hypercube Sampling

This class provides an implementation for a class for designing experiments to sample the parameter space of a complex model using Latin Hypercube sampling. The parameter space can be specified in a variety of ways, but essentially the user must provide a Probability Point Function (PPF, or inverse of the Cumulative Distribution Function) for each input parameter. Each PPF function takes a single numeric input and maps from the interval $[0, 1]$ to the desired parameter distribution value for a given parameter, and each parameter has a separate function describing its distribution. Note that this makes the assumption of no correlations between any of the parameter values (a future version may implement an experimental design where there are such parameter correlations). Once the design is initialized, a desired number of samples can be drawn from the design, returning an array holding the desired number of samples from the parameter space.

Internally, the class holds the set of PPFs for all of the parameter values, and samples are drawn by calling the `sample` method. To draw the samples, the `_draw_samples` is used to generate a series of points in the $[0, 1]^n$ hypercube using Latin Hypercube sampling, where n is the number of paramters. This set of samples from the Latin Hypercube is then mapped to the parameter space using the given PPF functions.

Unlike Monte Carlo sampling, Latin Hypercube designs attempt to sample more uniformly from the parameter space. Latin Hypercube sampling ensures that each sample is drawn from a different part of the space for each parameter. For example, if four samples are drawn, then for each parameter, one sample is guaranteed to be drawn from each quartile of the distribution. This ensures a more uniform sampling when compared on Monte Carlo sampling, but requires slightly more computation to generate the samples. Note however, that for very large numbers of parameters, Latin Hypercubes still may not sample very efficiently. This is due to the fact that the size of the parameter space grows exponentially with the number of dimensions, so a fixed number of samples will sample the space more poorly as the number of parameters increases.

class `mogp_emulator.ExperimentalDesign.`**LatinHypercubeDesign**(**args*)

Class representing a one-shot design of experiments with uncorrelated parameters using Latin Hypercube Sampling

This class provides an implementation for a class for designing experiments to sample the parameter space of a complex model using Latin Hypercube sampling. The parameter space can be specified in a variety of ways, but essentially the user must provide a Probability Point Function (PPF, or inverse of the Cumulative Distribution Function) for each input parameter. Each PPF function takes a single numeric input and maps from the interval $[0, 1]$ to the desired parameter distribution value for a given parameter, and each parameter has a separate function describing its distribution. Note that this makes the assumption of no correlations between any of the parameter values (a future version may implement an experimental design where there are such parameter correlations). Once the design is initialized, a desired number of samples can be drawn from the design, returning an array holding the desired number of samples from the parameter space.

Internally, the class holds the set of PPFs for all of the parameter values, and samples are drawn by calling the `sample` method. To draw the samples, the `_draw_samples` is used to generate a series of points in the $[0, 1]^n$ hypercube using Latin Hypercube sampling, where n is the number of paramters. This set of samples from the Latin Hypercube is then mapped to the parameter space using the given PPF functions.

Unlike Monte Carlo sampling, Latin Hypercube designs attempt to sample more uniformly from the parameter space. Latin Hypercube sampling ensures that each sample is drawn from a different part of the space for each parameter. For example, if four samples are drawn, then for each parameter, one sample is guaranteed to be drawn from each quartile of the distribution. This ensures a more uniform sampling when compared on Monte Carlo sampling, but requires slightly more computation to generate the samples. Note however, that for very large numbers of parameters, Latin Hypercubes still may not sample very efficiently. This is due to the fact that the size of the parameter space grows exponentially with the number of dimensions, so a fixed number of samples will sample the space more poorly as the number of parameters increases.

`__init__`(**args*)

Create a new instance of a Latin Hypercube experimental design

Creates a new instance of a Latin Hypercube design of experiments, which draws samples from the parameter space of a complex model in a more uniform fashion when compared to random Monte Carlo sampling. It can be used to generate data for a Gaussian Process emulator to fit the outputs of the complex model. Because the samples are drawn more uniformly than in Monte Carlo sampling, these designs may perform better in high dimensional parameter spaces.

The experimental design can be initialized in several ways depending on the arguments provided, ranging from the simplest to the most complicated.

1. Provide an integer `n` indicating the number of input parameters. If this is used to create an instance, it is assumed that all parameters are unformly distributed over the n-dimensional hypercube.
2. Provide an integer `n` and a tuple `(a, b)` of length 2 containing two numeric values (where $a < b$). In this case, all parameters are assumed to be uniformly distributed over the interval $[a, b]$.
3. Provide an integer `n` and a function that takes a single numeric input in the interval $[0, 1]$ and maps it to the parameter space. In this case, all parameters are assumed to follow the provided Probability Point Function.
4. Provide a list of tuples of length 2 containing numeric values (as above, the first number must smaller than the second number). The design then assumes that the number of parameters is the length of the list, and each parameter follows a uniform distribution with the bounds given by the respective tuple in the given list.
5. Provide a list of functions taking a single input (as above, each function must map the interval $[0, 1]$ to the parameter space). The number of parameters in the design is the length of the list, and the given PPF functions define the parameter space for each input.

More concretely, if one input parameter is given, you may initilize the class in any of the following ways:

Parameters **`n_parameters`** (*`int`*) – Integer specifying the number of parameters (must be positive). The design will sample each parameter over the interval $[0, 1]$

or

Parameters **bounds_list** (*list*) – List of tuples containing two numeric values, each of which has the smaller number first. Each parameter then takes a uniform distribution with bounds given by each tuple.

or

Parameters **ppf_list** (*list*) – List of functions or other callable, each of which accepts one argument and maps the interval $[0, 1]$ to the parameter space. Each parameter follows the distribution given by the respective PPF function.

and if two input parameters are given:

Parameters

- **n_parameters** (*int*) – Integer specifying the number of parameters (must be positive). The design will sample each parameter over the interval $[0, 1]$
- **bounds** (*tuple*) – Tuple or other iterable containing two numeric values, where the smaller number must come first. Each parameter then takes a uniform distribution with bounds given by the numbers provided in the tuple.

or

Parameters

- **n_parameters** (*int*) – Integer specifying the number of parameters (must be positive). The design will sample each parameter over the interval $[0, 1]$
- **ppf** (*function*) – Function or other callable, which accepts one argument and maps the interval $[0, 1]$ to the parameter space. Each parameter follows the distribution given by the PPF function.

The scipy.stats package provides implementations of a wide range of distributions, with pre-defined PPF functions. See the Scipy user manual for more details. Note that in order to get a compatible PPF function that only takes a single input, you will need to set any parameters needed to define the distribution.

Internally, the class defines any PPF functions based on the input data and collects all of the PPF functions in a list. The class also contains information on the method used to draw samples from the design.

get_method()

Returns the method used to draw samples from the design

This method returns the method used to draw samples from the experimental design. The base class does not implement a method, so if you try to call this on the base class the code will raise a NotImplementedError. When deriving new designs from the base class, the method should be set when calling the __init__ method.

Returns Method used to draw samples from the design.

Return type str

get_n_parameters()

Returns number of parameters in the experimental design

This method returns the number of parameters in the experimental design. This is set when initializing the object, an cannot be modified.

Returns Number of parameters in the experimental design.

Return type int

sample(*n_samples*)

Draw parameter samples from the experimental design

This method implements drawing parameter samples from the experimental design. The method does this by calling the `_draw_samples` method to obtain samples from the $[0,1]^n$ hypercube, where n is the number of parameters. The `sample``method then transforms these samples drawn from the low level method to the actual parameter values using the PPF functions provided when initilizing the object. Note that this method also checks that all parameter values are finite; if any ``NaN` values are returned, an error will be raised.

Note that by implementing the sampling in this way, modifications to the method to draw samples using a different protocol only needs to change the `_draw_samples` method. This makes it simpler to define new designs, as only a single method needs to be altered.

Parameters **n_samples** (*int*) – Number of samples to be drawn from the design (must be a positive integer)

Returns Samples drawn from the design parameter space as a numpy array with shape `(n_samples, n_parameters)`

Return type ndarray

9.13 The `SequentialDesign` Class

Base class representing a sequential experimental design

This class provides the base implementation of a class for designing experiments sequentially. This means that rather than picking all simulation points in a single step, the points are selected one by one, taking into account the information obtained by determining the true parameter value at each design point when selecting the next one. Sequential designs can be very useful when running expensive, high-dimensional simulations to ensure that a limited computational budget is used effectvely.

Instead of choosing all points at once, which is the case in a one-shot design, a sequential design does some additional computation work at each step to more carefully choose the next point. This means that sequential designs are better suited for very expensive simulations, where the additional cost of choosing the next point is small compared to the overall computational cost of running the simulations.

A sequential design is built on top of a base design (which must be a subclass of the `ExperimentalDesign` class. In addition to the base design, the class must contain information on how many points are used in the initial design (i.e. the number of starting points used before starting the sequential steps in the design) and the number of candidate points that are considered during each iteration. Optionally, a function for evaluating the actual simulation can be optionally bound to the class instance, which allows the entire design process to be automated. If such a function is not provided, then the steps to run the design must be carried out manually, with the evaluated simulation values provided to the class at the end of each simulation in order to determine the next point.

To use the base class to create an experimental design, a new subclass must be created that provides a method `_eval_metric`, which considers all candidate points and returns the index of the best candidate. Otherwise, all other code provided here allows for a generic sequential design to be easily run and managed.

class `mogp_emulator.SequentialDesign.`**`SequentialDesign`**(*base_design*, *f=None*, *n_samples=None*, *n_init=10*, *n_cand=50*)

Base class representing a sequential experimental design

This class provides the base implementation of a class for designing experiments sequentially. This means that rather than picking all simulation points in a single step, the points are selected one by one, taking into account the information obtained by determining the true parameter value at each design point when selecting the next one. Sequential designs can be very useful when running expensive, high-dimensional simulations to ensure that a limited computational budget is used effectvely.

Instead of choosing all points at once, which is the case in a one-shot design, a sequential design does some additional computation work at each step to more carefully choose the next point. This means that sequential designs are better suited for very expensive simulations, where the additional cost of choosing the next point is small compared to the overall computational cost of running the simulations.

A sequential design is built on top of a base design (which must be a subclass of the `ExperimentalDesign` class. In addition to the base design, the class must contain information on how many points are used in the initial design (i.e. the number of starting points used before starting the sequential steps in the design) and the number of candidate points that are considered during each iteration. Optionally, a function for evaluating the actual simulation can be optionally bound to the class instance, which allows the entire design process to be automated. If such a function is not provided, then the steps to run the design must be carried out manually, with the evaluated simulation values provided to the class at the end of each simulation in order to determine the next point.

To use the base class to create an experimental design, a new subclass must be created that provides a method `_eval_metric`, which considers all candidate points and returns the index of the best candidate. Otherwise, all other code provided here allows for a generic sequential design to be easily run and managed.

`__init__`(*base_design*, *f=None*, *n_samples=None*, *n_init=10*, *n_cand=50*)

Create a new instance of a sequential experimental design

Creates a new instance of a sequential experimental design, which sequentially chooses points to be evaluated from a complex simulation function. It is often used for expensive computational models, where the cost of running a single evaluation is large and must be done in series due to computational limitations, and thus some additional computation done at each step to select new points is small compared to the overall cost of running a single simulation.

Sequential designs require specifying a base design using a subclass of `ExperimentalDesign` as well as information on the number of points to use in each step in the design process. Additionally, the function to evaluated can be bound to the class to allow automatic evaluation of the function at each step.

Parameters

- **`base_design`** (`ExperimentalDesign`) – Base one-shot experimental design (must be a subclass of `ExperimentalDesign`). This contains the information on the parameter space to be sampled.
- **`f`** (*`function or other callable`*) – Function to be evaluated for the design. Must take all parameter values as a single input array and return a single float or an array of length 1
- **`n_samples`** (*`int or None`*) – Number of sequential design points to be drawn. If specified, this must be a non-negative integer. Note that this is in addition to the number of initial points, meaning that the total design size will be `n_samples + n_init`. This can also be specified when running the full design. This parameter is optional, and defaults to `None` (meaning the number of samples is set when running the design, or that samples will be added manually).
- **`n_init`** (*`int`*) – Number of points in the inital design before the sequential steps begin. Must be a positive integer. Optional, default value is 10.
- **`n_cand`** – Number of candidates to consider at each sequential design step. Must be a positive integer. Optional, default value is 50.

`generate_initial_design`()

Create initial design

Method to set the initial design inputs. Generates the desired number of points for the initial design by drawing from the base design. Method sets the `inputs` attribute of the `SequentialDesign` instance, but also returns the initial design as a numpy array if the simulations are to be run manually. This method can be run repeatedly to draw different initial designs if the initial target values have not been set, but once the targets have been set the method will not overwrite them to prevent corruption of the design.

Returns Initial design points, a 2D numpy array with shape `(n_init, n_parameters)`

Return type ndarray

`get_base_design``()`

Get type of base design

Returns the type of the base design. The base design must be a subclass of `ExperimentalDesign`, but any one-shot design method can be used to generate the initial design and the candidates.

Returns Base design type as a string

Return type str

`get_batch_points``(`*n_points*`)`

Batch version of get_next_point for a Sequential Design

This method returns a batch of design points to run from a Sequential Design. This is useful if simulations can be run in parallel, which speeds up the ability to generate designs efficiently. The method simply calls `get_next_point` the required number of times, but rather than using the true value of the simulation it instead substitutes the predicted value that is method-specific. This can be implemented in a subclass by defining the method `_estimate_next_target`.

Parameters **`n_points`** (*`int`*) – Size of batch to generate for the next set of simulation points. This parameter determines the shape of the output array. Must be a positive integer.

Returns Set of batch points chosen using the batch version of the design as a numpy array with shape `(n_points, n_parameters)`

Return type ndarray

`get_candidates``()`

Get current candidate design input points

Returns a numpy array holding the current candidate design points. The array is 2D and has shape `(n_cand, n_parameters)`. It always has the same size once it is initialized, but the values will change acros iterations as new candidate points are considered at each iteration.

Returns Current value of the candidate design inputs

Return type ndarray

`get_current_iteration``()`

Get number of current iteration in the experimental design

Returns the current iteration during the sequential design process. This is mostly useful if the sequential design is being updated manually to know the current iteration.

Returns Current iteration number

Return type int

`get_inputs``()`

Get current design input points

Returns a numpy array holding the current design points. The array is 2D and has shape `(current_iteration, n_parameters)` (i.e. it is resized after each iteration when a new design point is chosen).

Returns Current value of the design inputs

Return type ndarray

get_n_cand()

Get number of candidate design points

Returns the number of candidate design points used in each sequential design step. Candidates are redrawn at each step, so this number of points will be drawn each time and all points will be considered at each iteration.

Returns Number of candidate design points

Return type int

get_n_init()

Get number of initial design points

Returns the number of initial design points used before beginning the sequential design steps. Note that this means that the total number of samples to be drawn for the design is `n_init + n_samples`.

Returns Number of initial design points

Return type int

get_n_parameters()

Get number of parameters in design

Returns the number of parameters in the design (note that this is specified in the base design that must be provided when initializing the class instance).

Returns Number of parameters in the design

Return type int

get_n_samples()

Get number of sequential design points

Returns the number of sequential design points used in the sequential design steps. This parameter can be `None` to indicate that the number of samples will be specified when running the design, or that the samples will be updated manually. Note that the total number of samples to be drawn for the design is `n_init + n_samples`.

Returns Number of sequential design points

Return type int

get_next_point()

Evaluate candidates to determine next point

Public method for determining the next point in the design. Internally, it checks that the inputs and target arrays are as expected for correctly drawing a new point, generates prospective candidates, and then evaluates them using the desired metric in order to select the best one. It updates the `inputs` array and returns the next point to be evaluated as a 1D numpy array of length `n_parameters`.

Returns Next design point, a 1D numpy array of length `n_parameters`

Return type ndarray

get_targets()

Get current design target points

Returns a numpy array holding the current target points. The array is 1D and has shape `(current_iteration,)` (i.e. it is resized after each iteration when a new target point is added). Note that simulation outputs must be a single number, so if considering a simulation has multiple outputs, the user must decide how to combine them to form the relevant target value for deciding which point to simulate next.

Returns Current value of the target inputs

Return type ndarray

has_function()

Determines if class contains a function for running the simulator

This method checks to see if a function has been provided for running the simulation.

Returns Whether or not the design has a bound function for evaluting the simulation.

Return type bool

load_design(*filename*)

Load previously saved sequential design

Loads a previously saved sequential design from file. Loads the arrays for `inputs`, `targets`, and `candidates` from file and sets other internal data to be consistent. It performs a few checks for consistency to ensure that the loaded design is compatible with the selected parameters, however, it does not completely check everything for consistency (in particular, it does not make any attempt to ensure that the exact base design or function are identical to what was previously used). It is up to the user to ensure that these are consistent with the previous instance of the design.

Parameters **filename** (*str or file*) – Filename or file object from which the design will be loaded

Returns None

run_initial_design()

Run initial design

Method to run the initial design by generating the initial design, evaluating the function on all design points, and setting the target values. Note that this requires having a bound function to the class in order to evaluate the design points internally. It is a shortcut to running `generate_initial_design`, evaluating the initial design points, and then using `set_initial_targets` to set the target values, with some additional checks along the way.

If the initial design has already been fully run, this method will raise an error as the method to generate the initial design checks this prior to overwriting the initial targets. Note also that this method checks that the outputs of the bound function match up with the expected array sizes and that all outputs are finite before updating the initial targets.

Returns None

Return type None

run_next_point()

Perform one iteration of the sequential design process

Method for performing an iteration of the sequential design process. This is a shortcut for generating and evaluating the candidates to find the best next design point, evaluating the function on the next point, and then updating the targets array with the value. This requires a function be bound to the class instance to automatically run the simulation. This will also automatically update the `current_iteration` attribute, which can be used to determine the number of sequential design steps that have been run.

Returns None

Return type None

run_sequential_design(*n_samples=None*)

Run the entire sequential design

Method to run all steps of the sequential design process. Note that the class instance must have a bound function for evaluating the design points to run all steps automatically. If such a method is not provided, the design steps must be run manually.

The desired number of samples to be drawn can either be specified when initializing the class instance or when calling this method. If a number of samples is provided on both occasions, then the number provided when calling `run_sequential_design` is used.

Internally, this method is a wrapper to `run_initial_design` and then calling `run_next_point` a total of `n_samples` times. Note that this means that the total number of design points is `n_init + n_samples`.

Parameters **`n_samples`** (*`int or None`*) – Number of sequential design steps to be run. Optional if the number was specified upon initialization. Default is `None` (default to number set when initializing). If numbers are provided on both occasions, the number set here is used. If a number is provided, must be non-negative.

Returns None

Return type None

`save_design` (*filename*)

Save current state of the sequential design

Saves the current state of the sequential design by writing the current values of `inputs`, `targets`, and `candidates` to file as a `.npz` file. To re-load a saved design, use the `load_design` method.

Note that this method only dumps the arrays holding the inputs, targets, and candidates to a `.npz` file. It does not ensure that the function or base design are consistent, so it is up to the user to ensure that the new design parameters are the same as the parameters for the old one.

Parameters **`filename`** (*`str or file`*) – Filename or file object where design will be saved

Returns None

`set_batch_targets` (*new_targets*)

Batch version of set_next_target for a Sequential Design

This method updates the targets array for a batch set of simulations. The input array must have shape `(n_points,)`, where `n_points` is the number of points selected when calling `get_batch_points`. Disagreement between these two values will result in an error.

Parameters **`new_targets`** (*`ndarray`*) – Array holding results from the simulations. Must be an array of shape `(n_points,)`, where `n_points` is set when calling `get_batch_points`

Returns None

`set_initial_targets` (*targets*)

Set initial design target values

Method to set the initial design targets. Generates the desired number of points for the initial design by drawing from the base design. Method sets the `inputs` attribute of the `SequentialDesign` instance, but also returns the initial design as a numpy array if the simulations are to be run manually. This method can be run repeatedly to draw different initial designs if the initial target values have not been set, but once the targets have been set the method will not overwrite them to prevent corruption of the design.

Target values must be an array with length `(n_init,)`, with values obtained by running the initial design through the simulation. Note that this means the initial design must be created prior to running this method – if this method is called prior to `generate_initial_design`, the code will raise an error.

Parameters **`targets`** (*`ndarray`*) – Initial value of targets, must be a 1D numpy array with shape `(n_init,)`

Returns None

Return type None

set_next_target(*target*)
Set value of next target

Updates the target array with the correct value (from running the actual simulation) of the latest design point determined using `get_next_point`. The target input must be a float or an array of length 1. The code internally checks the inputs and targets for any problems that may have occurred in updating them correctly, and if all is well then updates the target array and increments the number of iterations. If the design has not been correctly initialized, or `get_next_point` has not been previously run, this method will raise an error.

Parameters **target** (*float or length 1 array*) – New target value found from evaluating the simulation on the latest design point found from the `get_next_point` method.

Returns None

Return type None

9.14 The `MICEDesign` Class

Class representing a Mutual Information for Computer Experiments (MICE) sequential experimental design

This class provides an implementation of the MICE algorithm, which uses Mutual Information as the criterion for selecting new points in a sequential design. The idea in MICE is to select design points based on the point that provides the most information on the function values in the entire design space. This is a straightforward application of a sequential design procedure, though the class requires a few additional parameters in order to compute the MICE criteria.

These additional parameters are nugget parameters provided to the Gaussian Process fit to smooth the predictions when evaluating the Mutual Information criteria. Essentially, since experimental design often requires sampling from a high dimensional space, this cannot be done in a way that guarantees that all candidate points are equally spaced. The Mutual Information criterion is sensitive to how these candidate points are distributed in space, so the nugget parameter provides some smoothing that makes the criterion less dependent on the distribution of the candidate points. Typical values of the smoothing nugget parameters (`nugget_s` in this implementation) are 1, though this may depend on the application.

Other than the smoothing parameters, the implementation follows the base procedure for a sequential design. The implementation adds methods for querying the nugget parameters and an additional helper function for computing the Mutual Information criterion, but other methods are identical.

class mogp_emulator.SequentialDesign.**MICEDesign**(*base_design*, *f=None*, *n_samples=None*, *n_init=10*, *n_cand=50*, *nugget='adaptive'*, *nugget_s=1.0*)
Class representing a Mutual Information for Computer Experiments (MICE) sequential experimental design

This class provides an implementation of the MICE algorithm, which uses Mutual Information as the criterion for selecting new points in a sequential design. The idea in MICE is to select design points based on the point that provides the most information on the function values in the entire design space. This is a straightforward application of a sequential design procedure, though the class requires a few additional parameters in order to compute the MICE criteria.

These additional parameters are nugget parameters provided to the Gaussian Process fit to smooth the predictions when evaluating the Mutual Information criteria. Essentially, since experimental design often requires sampling from a high dimensional space, this cannot be done in a way that guarantees that all candidate points are equally spaced. The Mutual Information criterion is sensitive to how these candidate points are distributed in space, so the nugget parameter provides some smoothing that makes the criterion less dependent on the distribution of the candidate points. Typical values of the smoothing nugget parameters (`nugget_s` in this implementation) are 1, though this may depend on the application.

Other than the smoothing parameters, the implementation follows the base procedure for a sequential design. The implementation adds methods for querying the nugget parameters and an additional helper function for computing the Mutual Information criterion, but other methods are identical.

`__init__`(*base_design*, *f=None*, *n_samples=None*, *n_init=10*, *n_cand=50*, *nugget='adaptive'*, *nugget_s=1.0*)

Create new instance of a MICE sequential design

Method to initialize a new MICE design. Parameters are largely the same as for the base `SequentialDesign` class, with a few additional nugget parameters for computing the Mutual Information criterion. A base design must be provided (must be a subclass of the `ExperimentalDesign` class), plus optionally a function to be evaluated in the design. Additional parameters include the number of samples, the number of initial design points, the number of candidate points, the nugget parameter for the base GP, and the smoothing nugget parameter for smoothing the uncertainty predictions on the candidate design points. Note that the total number of design points is `n_init + n_samples`.

Parameters

- **`base_design`** (`ExperimentalDesign`) – Base one-shot experimental design (must be a subclass of `ExperimentalDesign`). This contains the information on the parameter space to be sampled.
- **`f`** (*`function or other callable`*) – Function to be evaluated for the design. Must take all parameter values as a single input array and return a single float or an array of length 1
- **`n_samples`** (*`int or None`*) – Number of sequential design points to be drawn. If specified, this must be a positive integer. Note that this is in addition to the number of initial points, meaning that the total design size will be `n_samples + n_init`. This can also be specified when running the full design. This parameter is optional, and defaults to `None` (meaning the number of samples is set when running the design, or that samples will be added manually).
- **`n_init`** (*`int`*) – Number of points in the inital design before the sequential steps begin. Must be a positive integer. Optional, default value is 10.
- **`n_cand`** – Number of candidates to consider at each sequential design step. Must be a positive integer. Optional, default value is 50.
- **`nugget`** (*`float or None`*) – Nugget parameter for base GP predictions. Must be a non-negative float or `None`, where `None` indicates that the nugget parameter is selected adaptively. Optional, default value is `None`.
- **`nugget_s`** (*`float`*) – Smoothing nugget parameter for smoothing the predictions on the candidate space. Must be a non-negative float. Default value is 1.

`generate_initial_design`()

Create initial design

Method to set the initial design inputs. Generates the desired number of points for the initial design by drawing from the base design. Method sets the `inputs` attribute of the `SequentialDesign` instance, but also returns the initial design as a numpy array if the simulations are to be run manually. This method can be run repeatedly to draw different initial designs if the initial target values have not been set, but once the targets have been set the method will not overwrite them to prevent corruption of the design.

Returns Initial design points, a 2D numpy array with shape `(n_init, n_parameters)`

Return type ndarray

get_base_design()

Get type of base design

Returns the type of the base design. The base design must be a subclass of `ExperimentalDesign`, but any one-shot design method can be used to generate the initial design and the candidates.

Returns Base design type as a string

Return type str

get_batch_points(*n_points*)

Batch version of get_next_point for a Sequential Design

This method returns a batch of design points to run from a Sequential Design. This is useful if simulations can be run in parallel, which speeds up the ability to generate designs efficiently. The method simply calls `get_next_point` the required number of times, but rather than using the true value of the simulation it instead substitutes the predicted value that is method-specific. This can be implemented in a subclass by defining the method `_estimate_next_target`.

Parameters **n_points** (*int*) – Size of batch to generate for the next set of simulation points. This parameter determines the shape of the output array. Must be a positive integer.

Returns Set of batch points chosen using the batch version of the design as a numpy array with shape `(n_points, n_parameters)`

Return type ndarray

get_candidates()

Get current candidate design input points

Returns a numpy array holding the current candidate design points. The array is 2D and has shape `(n_cand, n_parameters)`. It always has the same size once it is initialized, but the values will change acros iterations as new candidate points are considered at each iteration.

Returns Current value of the candidate design inputs

Return type ndarray

get_current_iteration()

Get number of current iteration in the experimental design

Returns the current iteration during the sequential design process. This is mostly useful if the sequential design is being updated manually to know the current iteration.

Returns Current iteration number

Return type int

get_inputs()

Get current design input points

Returns a numpy array holding the current design points. The array is 2D and has shape `(current_iteration, n_parameters)` (i.e. it is resized after each iteration when a new design point is chosen).

Returns Current value of the design inputs

Return type ndarray

get_n_cand()

Get number of candidate design points

Returns the number of candidate design points used in each sequential design step. Candidates are re-drawn at each step, so this number of points will be drawn each time and all points will be considered at each iteration.

Returns Number of candidate design points

Return type int

get_n_init()

Get number of initial design points

Returns the number of initial design points used before beginning the sequential design steps. Note that this means that the total number of samples to be drawn for the design is `n_init + n_samples`.

Returns Number of initial design points

Return type int

get_n_parameters()

Get number of parameters in design

Returns the number of parameters in the design (note that this is specified in the base design that must be provided when initializing the class instance).

Returns Number of parameters in the design

Return type int

get_n_samples()

Get number of sequential design points

Returns the number of sequential design points used in the sequential design steps. This parameter can be `None` to indicate that the number of samples will be specified when running the design, or that the samples will be updated manually. Note that the total number of samples to be drawn for the design is `n_init + n_samples`.

Returns Number of sequential design points

Return type int

get_next_point()

Evaluate candidates to determine next point

Public method for determining the next point in the design. Internally, it checks that the inputs and target arrays are as expected for correctly drawing a new point, generates prospective candidates, and then evaluates them using the desired metric in order to select the best one. It updates the `inputs` array and returns the next point to be evaluated as a 1D numpy array of length `n_parameters`.

Returns Next design point, a 1D numpy array of length `n_parameters`

Return type ndarray

get_nugget()

Get value of nugget parameter for base GP

Returns the nugget value for the base GP (used to actually fit the inputs to targets). Can be a float or None (meaning fitting will adaptively add noise to stabilize matrix inversion as needed).

Returns Nugget parameter, can be a float or None for adaptive noise addition.

Return type float or None

get_nugget_s()

Get value of smoothing nugget parameter

Returns the value of the smoothing nugget parameter for the GP used to evaluate the mutual information criterion. This GP examines the correlation between a candidate design point and the other candidate points, which requires smoothing to ensure that the correlation measure is not biased by the distribution of the candidate points in space. This parameter must be a nonnegative float (typical values used are 1, though this may depend on the application).

Returns Nugget parameter for smoothing predictions from candidate points made on a candidate point. Typical values are 1.

Return type float

get_targets()

Get current design target points

Returns a numpy array holding the current target points. The array is 1D and has shape `(current_iteration,)` (i.e. it is resized after each iteration when a new target point is added). Note that simulation outputs must be a single number, so if considering a simulation has multiple outputs, the user must decide how to combine them to form the relevant target value for deciding which point to simulate next.

Returns Current value of the target inputs

Return type ndarray

has_function()

Determines if class contains a function for running the simulator

This method checks to see if a function has been provided for running the simulation.

Returns Whether or not the design has a bound function for evaluting the simulation.

Return type bool

load_design(*filename*)

Load previously saved sequential design

Loads a previously saved sequential design from file. Loads the arrays for `inputs`, `targets`, and `candidates` from file and sets other internal data to be consistent. It performs a few checks for consistency to ensure that the loaded design is compatible with the selected parameters, however, it does not completely check everything for consistency (in particular, it does not make any attempt to ensure that the exact base design or function are identical to what was previously used). It is up to the user to ensure that these are consistent with the previous instance of the design.

Parameters **filename** (*str or file*) – Filename or file object from which the design will be loaded

Returns None

run_initial_design()

Run initial design

Method to run the initial design by generating the initial design, evaluating the function on all design points, and setting the target values. Note that this requires having a bound function to the class in order to evaluate the design points internally. It is a shortcut to running `generate_initial_design`, evaluating the initial design points, and then using `set_initial_targets` to set the target values, with some additional checks along the way.

If the initial design has already been fully run, this method will raise an error as the method to generate the initial design checks this prior to overwriting the initial targets. Note also that this method checks that the outputs of the bound function match up with the expected array sizes and that all outputs are finite before updating the initial targets.

Returns None

Return type None

run_next_point()

Perform one iteration of the sequential design process

Method for performing an iteration of the sequential design process. This is a shortcut for generating and evaluating the candidates to find the best next design point, evaluating the function on the next point, and then updating the targets array with the value. This requires a function be bound to the class instance to automatically run the simulation. This will also automatically update the `current_iteration` attribute, which can be used to determine the number of sequential design steps that have been run.

Returns None

Return type None

run_sequential_design(*n_samples=None*)

Run the entire sequential design

Method to run all steps of the sequential design process. Note that the class instance must have a bound function for evaluating the design points to run all steps automatically. If such a method is not provided, the design steps must be run manually.

The desired number of samples to be drawn can either be specified when initializing the class instance or when calling this method. If a number of samples is provided on both occasions, then the number provided when calling `run_sequential_design` is used.

Internally, this method is a wrapper to `run_initial_design` and then calling `run_next_point` a total of `n_samples` times. Note that this means that the total number of design points is `n_init + n_samples`.

Parameters **n_samples** (*int or None*) – Number of sequential design steps to be run. Optional if the number was specified upon initialization. Default is `None` (default to number set when initializing). If numbers are provided on both occasions, the number set here is used. If a number is provided, must be non-negative.

Returns None

Return type None

save_design(*filename*)

Save current state of the sequential design

Saves the current state of the sequential design by writing the current values of `inputs`, `targets`, and `candidates` to file as a `.npz` file. To re-load a saved design, use the `load_design` method.

Note that this method only dumps the arrays holding the inputs, targets, and candidates to a `.npz` file. It does not ensure that the function or base design are consistent, so it is up to the user to ensure that the new design parameters are the same as the parameters for the old one.

Parameters **filename** (*str or file*) – Filename or file object where design will be saved

Returns None

set_batch_targets(*new_targets*)

Batch version of set_next_target for a Sequential Design

This method updates the targets array for a batch set of simulations. The input array must have shape `(n_points,)`, where `n_points` is the number of points selected when calling `get_batch_points`. Disagreement between these two values will result in an error.

Parameters **new_targets** (*ndarray*) – Array holding results from the simulations. Must be an array of shape `(n_points,)`, where `n_points` is set when calling `get_batch_points`

Returns None

set_initial_targets(*targets*)

Set initial design target values

Method to set the initial design targets. Generates the desired number of points for the initial design by drawing from the base design. Method sets the `inputs` attribute of the `SequentialDesign` instance, but also returns the initial design as a numpy array if the simulations are to be run manually. This method can be run repeatedly to draw different initial designs if the initial target values have not been set, but once the targets have been set the method will not overwrite them to prevent corruption of the design.

Target values must be an array with length `(n_init,)`, with values obtained by running the initial design through the simulation. Note that this means the initial design must be created prior to running this method – if this method is called prior to `generate_initial_design`, the code will raise an error.

Parameters **targets** (*ndarray*) – Initial value of targets, must be a 1D numpy array with shape `(n_init,)`

Returns None

Return type None

set_next_target(*target*)

Set value of next target

Updates the target array with the correct value (from running the actual simulation) of the latest design point determined using `get_next_point`. The target input must be a float or an array of length 1. The code internally checks the inputs and targets for any problems that may have occurred in updating them correctly, and if all is well then updates the target array and increments the number of iterations. If the design has not been correctly initialized, or `get_next_point` has not been previously run, this method will raise an error.

Parameters **target** (*float or length 1 array*) – New target value found from evaluating the simulation on the latest design point found from the `get_next_point` method.

Returns None

Return type None

9.15 The `MICEFastGP` Class

Derived GaussianProcess class implementing the Woodbury matrix identity for fast predictions

This class implements a Gaussian Process that is used in the MICE Sequential Design. The GP is fit using all candidate points from the sequential design, and the uses the Woodbury matrix identity to correct that fit to exclude the candidate point in question. This reduces the cost of fitting the GP from O(n^3) to O(n^2), which can dramatically speed up this process for large numbers of candidate points. This is mostly used for the particular application to the MICE sequential design, but could potentially have other applications where many candidate points are to be considered one at a time.

class mogp_emulator.SequentialDesign.**MICEFastGP**(*inputs*, *targets*, *mean=None*, *kernel=<mogp_emulator.Kernel.SquaredExponential object>*, *priors=None*, *nugget='adaptive'*, *inputdict={}*, *use_patsy=True*)

Derived GaussianProcess class implementing the Woodbury matrix identity for fast predictions

This class implements a Gaussian Process that is used in the MICE Sequential Design. The GP is fit using all candidate points from the sequential design, and the uses the Woodbury matrix identity to correct that fit to exclude the candidate point in question. This reduces the cost of fitting the GP from O(n^3) to O(n^2), which can dramatically speed up this process for large numbers of candidate points. This is mostly used for the particular application to the MICE sequential design, but could potentially have other applications where many candidate points are to be considered one at a time.

fast_predict (*index*)

Make a fast prediction using one input point to a fit GP

This method is used to correct a Gaussian Process fit to a set of candidate points to evaluate the uncertainty at the candidate point. It is used in the MICE sequential design procedure to examine the mutual information between candidate points by determining how well correlated the design point is in question to the remainder of the candidates. It uses the Woodbury matrix identity to correct the existing GP fit (which requires O(n^3) operations) using O(n^2) operations, speeding up the process significantly for large candidate design sizes.

The method requires a fit GP, and the index of the input point that is to be excluded. The method then corrects the GP fit and computes the uncertainty of the prediction on the excluded point returning the uncertainty as a float.

Parameters **index** (*int*) – Index of input point to be excluded in the fit and to which the prediction will be applied. Must be an integer with 0 <= index < n (where n is the number of target points in the fit GP, or the number of candidate points when applied to the MICE procedure).

Returns Uncertainty in the corrected fit applied to the given index point

Return type float

9.16 The HistoryMatching Class

class mogp_emulator.**HistoryMatching** (*gp=None*, *obs=None*, *coords=None*, *expectations=None*, *threshold=3.0*)

Class containing tools to implement history matching between the outputs of a gaussian process and an observation.

Primary usage is the `get_implausibility` method, which calculates the implausibility metric (a number of standard deviations between the emulator mean and the observation) for a number of query points:

$$I_i(\bar{x_0}) = \frac{|z_i - E(f_i(\bar{x_0}))|}{\sqrt{Var\left[z_i - E(f_i(\bar{x_0}))\right]}}$$

The code allows a number of ways to specify variances, and all are summed to compute the final variance that determines the implausibility measure. Variances on the observation itself can be provided with the observations, while variances arising from the GP prediction are included in the emulator expectations. Additional variances can be included to account for model discrepancy when computing the implausibility metric, and these can be uniform for all query points, or vary across different query points if desired. Any number of additional variances can be included in the calculation by passing them when computing the implausibility.

Once implausibility is computed, the class can determine query points that can be ruled out, as well as points that are not ruled out yet, based on the threshold class attribute. See the methods `get_RO`, `get_NROY`, and `set_threshold` for more details.

Example - implausibility computation for a 1D GP:

```
import math
import numpy as np
from HistoryMatching import HistoryMatching
from mogp_emulator import GaussianProcess

```

(continues on next page)

(continued from previous page)

```
# Set up a gaussian process
x_training = np.array([[0.],[10.],[20.],[30.],[43.],[50.]])
def get_y_training(x):
    n_points = len(x)
    f = np.zeros(n_points)
    for i in range(n_points):
        f[i] = np.sin(2*math.pi*x[i] / 50)
    return f
y_training = get_y_training(x_training)

gp = GaussianProcess(x_training, y_training)
np.random.seed(47)
gp.learn_hyperparameters()

# Define the observation to which to compare
obs = [-0.8, 0.0004]

# Coordinates to predict
n_rand = 2000
coords = np.sort(np.random.rand(n_rand),axis=0)*(56)-3
coords = coords[:,None]

# Generate Expectations
expectations = gp.predict(coords)

# Calculate implausibility
hm = HistoryMatching(obs=obs, expectations=expectations)
I = hm.get_implausibility()
```

`__init__`(*gp=None*, *obs=None*, *coords=None*, *expectations=None*, *threshold=3.0*)
Create a new instance of history matching.

The primary function of this method is to initialise the `self.gp`, `.obs`, `.coords`, `.expectations`, `.ndim`, `.ncoords`, `.threshold`, `.I`, `.NROY`, and `.RO` variables as placeholders. The optional keyword arguments can also be used to set user-defined values for various quantities in a single call to `HistoryMatching()`. Setting sufficient numbers of these to allow the number of dimensions and/or number of coordinates to be computed also causes these to be set at this stage.

Parameters

- **gp** (`GaussianProcess` *or None*) – (Optional) `GaussianProcess` object used to make predictions in history matching. Optional, can instead provide predictions directly via the `expectations` argument. Default is `None`
- **obs** (*float, list, or None*) – (Optional) Observations against which the predictions will be compared. If provided, must either be a float (assumes no uncertainty in the observations) or a list of two floats holding the observation and its variance. Required for history matching, but can be provided when calling `get_implausibility`. Default is `None`.
- **coords** (*ndarray*) – (Optional) Inputs at which the emulator values will be computed to compare the GP output to the observations. If provided, must be a numpy array matching the input for the GP (must be a 2D array with dimensions `(n, D)` where `n` is the number of points to be considered and `D` is the number of parameters for the GP). Only required if `expectations` is not provided. Default is `None`.
- **expectations** (*tuple holding 3 ndarrays*) – (Optional) Tuple of 3 numpy arrays or 2 numpy arrays and `None` of the form expected from GP predictions. The first

array must hold the predicted mean at all query points, and the second array must hold the variances from the GP predictions at all query points. The third is not used, so can simply be a dummy array or `None`. Can instead provide a GP object and the points to query to have the predictions made internally. Default is `None`.

`check_coords`(*coords*)

Checks if the provided argument is consistent with expectations for coordinates.

Returns a boolean that is `True` if the provided quantity is consistent with the requirements for the coordinates quantity, i.e. a ndarray of fewer than 3 dimensions.

Parameters **`coords`** (*`ndarray`*) – Input to check for consistency with coordinates.

Returns Boolean indicating if coords is consistent

Return type bool

`check_expectations`(*expectations*)

Checks if the provided argument is consistent with expectations for Gaussian Process Expectations.

Returns a boolean that is `True` if the provided quantity is consistent with the output of the predict method of a GaussianProcess object, i.e. that it is a `GaussianProcess.PredictResult` object with mean and variance defined.

Parameters **`expectations`** (*`tuple of 3 numpy arrays or 2 numpy arrays and None`*) – Input to check for consistency with expectations

Returns Boolean indicating if expectations is consistent

Return type bool

`check_gp`(*gp*)

Checks if the provided argument is consistent with expectations for a GP.

Returns a boolean that is True if the provided quantity is consistent with the requirements for a gaussian process, i.e. is of type `GaussianProcess`.

Parameters **`gp`** (`GaussianProcess`) – Input GP object to be checked.

Returns Boolean indicating if provided object is a `GaussianProcess`

Return type bool

`check_obs`(*obs*)

Checks if the provided argument is consistent with expectations for observations.

Returns a boolean that is `True` if the provided quantity is consistent with the requirements for the observation quantity, i.e. is a numerical value that can be converted to a float, or a list of up to two such values. Also checks if the provided variance is non-negative.

Parameters **`obs`** (*`float or list-like`*) – Input for observations to be checked

Returns Boolean indicating if provided observations are acceptable

Return type bool

`check_threshold`(*threshold*)

Check value of threshold

Checks if the provided argument is consistent with expectations for a threshold value.

Returns a boolean that is `True` if the provided quantity is consistent with the requirements for a threshold value, i.e. that it is a non-negative numerical value.

Parameters **`threshold`** (*`float`*) – threshold to be tested

Returns Boolean indicating if the provided argument is consistent with a threshold

Return type bool

get_NROY(*discrepancy=0.0*)

Return set of indices that are not yet ruled out

Returns a list of indices for `self.I` that correspond to entries that are not yet ruled out. Points that are ruled out have an implausibility metric that exceeds the threshold (can be set when initializing history matching or using the `set_threshold` method). If the implausibility metric has not yet been computed for the desired points, it is calculated. If a model discrepancy is to be included, it can be passed here.

Parameters **discrepancy** (*float*) – Additional variance to be included in the implausibility calculation. Must be a non-negative float. Optional, default is `0.`

Returns List of integer indices that have not yet been ruled out.

Return type list

get_RO(*discrepancy=0.0*)

Return set of indices that have been ruled out

Returns a list of indices for `self.I` that correspond to entries that have been ruled out. Points that are ruled out have an implausibility metric that exceeds the threshold (can be set when initializing history matching or using the `set_threshold` method). If the implausibility metric has not yet been computed for the desired points, it is calculated. If a model discrepancy is to be included, it can be passed here.

Parameters **discrepancy** (*float*) – Additional variance to be included in the implausibility calculation. Must be a non-negative float. Optional, default is `0.`

Returns List of integer indices that have been ruled out.

Return type list

get_implausibility(*discrepancy=0.0*)

Compute Implausibility measure for all query points

Carries out the implausibility calculation given by:

$$I_i(\bar{x_0}) = \frac{|z_i - E(f_i(\bar{x_0}))|}{\sqrt{Var\left[z_i - E(f_i(\bar{x_0}))\right]}}$$

to return an implausibility value (a number of standard deviations between the emulator mean and the observation)for each of the provided coordinates.

Requires that the observation parameter is set, and that at least one of the following conditions are met:

a) The coords and gp parameters are set

b) The expectations parameter is set

Note that using the GP internally assumes that the standard prediction settings will be used when making predictions. If the user wishes to have more control over the prediction method (i.e. make the predictions from MCMC samples), the user should explicitly pass `expectations` to the object.

An additional variance can be included that represents a general model discrepancy that describes the prior beliefs regarding how well the model matches reality. In practice, the model discrepancy is essential to have predictions that are not overly confident, however it can be hard to estimate (see further discussion in Brynjarsdottir and O'Hagan, 2014).

As the implausibility calculation linearly sums variances, the result is agnostic to the precise provenance of any of the included variances

Parameters **discrepancy** (*float*) – Additional variance to be included in the implausibility calculation. Must be a non-negative float. Optional, default is `0.`

Returns Array holding implausibility metric for all query points accounting for all variances, ndarray of length `(ncoords,)`

Return type ndarray

set_coords(*coords*)

Set the query points to be used in history matching

Sets the `self.coords` variable to the provided query points `coords` if it passes the `check_coords` requirements, or be `None` to remove a set of existing query points. `coords` must be a numpy array matching the inputs to the provided `GaussianProcess` object.

Parameters **coords** (*ndarray or None*) – Numpy array holding query points (array with shape `(n, D)`, where `n` is the number of query points and `D` is the number of inputs to the emulator), or `None`

Returns None

set_expectations(*expectations*)

Set the expected output of the simulator to be used in history matching

Sets the `self.expectations` variable to the provided `expectations` argument if it passes the `check_expectations` requirements, or `None` can be used to remove an existing set of expectations. Expectations must be a tuple of 3 numpy arrays, where the first holds the predicted means for all query points and the second holds the predicted variances for all query points. The third arrray is not used in the computation, but is included as it is an expected output of the GP `predict` method.

Parameters **expectations** (*tuple of 3 ndarrays or None*) – GP predictions at all query points. Must be a tuple of 3 numpy arrays, or None to remove existing expectations.

Returns None

set_gp(*gp*)

Set the Gaussian Process to use with history matching

Sets the `self.gp` variable to the provided `GaussianProcess` argument.

Parameters **gp** (`GaussianProcess`) – `GaussianProcess` object to use for history matching.

Returns None

set_obs(*obs*)

Set the observations to be used for history matching

Sets the `self.obs` variable to the provided `obs` argument. The object must pass the `check_obs` requirements, meaning that it must be a float (assumes that the observation has no error associated with it) or a list containing two floats (representing an observation and its variance). Note that the variance must be non-negative.

Parameters **obs** (*float or list*) – Observations to be used for history matching. Must be a float or a list of two floats.

Returns None

set_threshold(*threshold*)

Set the threshold value for history matching

Sets the `self.threshold variable` to the provided threshold argument if it passes the `check_threshold` requirements. The threshold must be a non-negative float.

Parameters **threshold** (*float*) – New value for threshold, must be a non-negative float.

Returns None

status()

Prints a summary of the current status of the class object.

Prints a summary of the current status of the class object, including the values stored in `self.gp`, `.obs`, `.ndim`, `.ncoords`, and `.threshold`, and the shapes/sizes ofvalues stored in `self.obs`, `.coords`, `.expectations`, `.I`, `.NROY`, and `.RO`. No inputs, no return value.

Returns None

update()

Update History Matching object

Checks that sufficient information exists to compute `ndim` and/or `ncoords` and, if so, computes these values. This also serves to update these values if the information on which they are based is changed. No inputs, no return value.

Returns None

9.17 The MCMC Module

`mogp_emulator.MCMC.`**MCMC_step**(*loglikelihood*, *current_params*, *step_sizes*, *loglike_sign=1.0*)

Method to take a weak prior Metropolis-Hastings MCMC step

Take a Metropolis-Hastings MCMC step with given log-likelihood function and step sizes. Method uses a multivariate normal distribution centered around the current parameter values from which the next point is drawn, and evaluates the log-likelihood for both points. If the next point has a larger log-likelihood, the step is always accepted, and if the log-likelihood is less for the proposed step it is accepted with a probability based on the difference between the two. If the method encounters an error when evalutaing the log-likelihood, the step is rejected. Returns the next point and a boolean indicating whether or not the step was accepted.

An optional parameter `loglike_sign` can be passed that must be a float with the value +/- 1. This is multiplied by the log-likelihood and thus allows methods that compute the negative log-likelihood to be used in this routine. If values other than +/- 1 are passed, the method will raise an error.

Parameters

- **loglikelihood** (*function or other callable*) – Log-likelihood function to be used in the MCMC step. Must be callable and must accept a single argument, which is the array holding the parameters. If this function computes the negative log-likelihood, pass `loglike_sign = -1.` to the function as well.
- **current_params** (*ndarray*) – Current value of the parameters. Must be a 1D array.
- **step_sizes** (*ndarray*) – Covariance matrix from which steps are drawn. Must be a 2D array with both dimensions the same length as `current_params`, and must be positive definite.
- **loglike_sign** (*float*) – Sign for the log-likelihood function. If the provided `loglikelihood` function computes the negative log-likelihood, pass `-1.` for this parameter. Optional, default value is `1.`

Returns Proposed next point and whether or not the point is accepted as a tuple. The first return item is the next point (as a 1D array with the same length as `current_params`) and the second item is a boolean indicating whether or not the step was accepted.

Return type tuple containing a ndarray and a bool mogp_emulator.MCMC.**MH_proposal**(*current_params*, *step_sizes*)

Propose an MCMC step using a Metropolis-Hastings method

Proposes the next point in an MCMC sampler using the Metropolis-Hastings method. Inputs are the current point and a covariance matrix. The next point is drawn from a multivariate normal distribution centered around the current point. The covariance matrix must be a 2D `n` by `n` array, where `n` is the number of parameters, and must be positive definite. Returns a 1D array of length `n` holding the new proposed parameter values.

Parameters

- **current_params** (*ndarray*) – Current value of the parameters. Must be a 1D array.
- **step_sizes** (*ndarray*) – Covariance matrix from which steps are drawn. Must be a 2D array with both dimensions the same length as `current_params`, and must be positive definite.

Returns New value of parameters, a 1D array with the same length as `current_params`

Return type ndarray

mogp_emulator.MCMC.**autothin_samples**(*signal*)

Automatically estimate thinning needed to obtain uncorrelated samples

This function attempts to estimate the thinning needed to obtain uncorrelated samples in an MCMC chain. For each separate parameter, the function computes the autocorrelation and estimates the lag needed to obtain uncorrelated samples. This is done by recognizing that the standard deviation of the autocorrelation of a random signal scales inversely with the square root of the number of samples, so when the autocorrelation drops below three times this, we use this as a guess to when the signal is uncorrelated. The maximum lag across all parameters is returned. If the chain contains fewer than 10 points, or the autocorrelation never drops below the target value, the method gives a warning and returns 1.

Parameters **signal** (*ndarray*) – MCMC chain to be thinned. Must be a 1D or 2D array. If 2D, the first dimension indicates the MCMC samples, and the second dimension indicates the different parameter values. If 1D, the array is assumed to hold samples of a single parameter, and the array is reshaped to have a singleton second dimension.

Returns Estimated stride needed to thin the samples to obtain uncorrelated samples. If the method does not succeed, gives a warning and returns `1`.

Return type int

mogp_emulator.MCMC.**sample_MCMC**(*loglikelihood*, *start*, *step_sizes*, *n_samples=1000*, *thin=0*, *loglike_sign=1.0*)

Draw MCMC samples for a given log-likelihood function with weak priors

Compute an MCMC chain for a given log-likelihood function with weak priors. Function requires the log-likelihood function, the starting point for the MCMC chain, and an array describing the step sizes. Optional parameters are the number of steps to take, how the thin the samples, and a sign for the log-likelihood function.

The log-likelihood function must be a function or other callable that accepts a single argument, which is a 1D array holding the current parameter values. The starting point must be a 1D array holding the starting parameter values, which must match the length of the input to the log-likelihood function. The step sizes must be a 2D array with each dimension having the same length as the number of parameters, and must be positive definite. The step size array is used as the covariance matrix for a multivariate normal distribution, from which steps are drawn.

Optional parameters are the number of steps to be taken (must be a positive integer). Note that if the chain is thinned, the number of points in the final MCMC chain will differ from the number of steps taken.

Thinning may be specified with a non-negative integer. If a positive integer is given, the chain will be thinned by only keeping every `thin` steps. Note that `thin = 1` means that the chain will not be thinned. If `thin = 0` is given (the default value), the chain will automatically be thinned by computing the autocorrelation of the chain for each parameter separately and estimating the value needed to eliminate correlations in the chain. If the autothinning method fails (usually occurrs if the posterior is multimodal), the chain will not be thinned and a warning will be given. More details on the autothinning procedure are described in the corresponding function.

An optional parameter `loglike_sign` can be passed that must be a float with the value +/- 1. This is multiplied by the log-likelihood and thus allows methods that compute the negative log-likelihood to be used in this routine.

Returns the final thinned MCMC chain (a 2D array, where the first dimension indicates the different samples and the second dimension indicates the different parameters), an array holding all rejected steps (also a 2D array like the MCMC chain, useful for diagnosing problems with convergence), the fraction of steps that are accepted (also useful for diagnosing problems with convergence), and the first lag autocorrelation of the thinned MCMC chain.

Parameters

- **`loglikelihood`** (*`function or other callable`*) – Log-likelihood function to be used in the MCMC step. Must be callable and must accept a single argument, which is the array holding the parameters. If this function computes the negative log-likelihood, pass `loglike_sign = -1.` to the function as well.
- **`start`** (*`ndarray`*) – Starting value of the parameters. Must be a 1D array.
- **`step_sizes`** (*`ndarray`*) – Covariance matrix from which steps are drawn. Must be a 2D array with both dimensions the same length as `current_params`, and must be positive definite.
- **`n_samples`** (*`int`*) – Number of steps to be taken. Must be a positive integer. Optional, default value is 1000. Note that if the chain is thinned, the final MCMC chain will be shorter than this.
- **`thin`** (*`int`*) – Integer describing how to thin the MCMC chain. A positive integer indicates manual thinning by keeping every `thin` steps (note that `thin = 1` means the chain will not be thinned). A value of `0` will attempt to autothin the chain.
- **`loglike_sign`** (*`float`*) – Sign for the log-likelihood function. If the provided `loglikelihood` function computes the negative log-likelihood, pass `-1.` for this parameter. Optional, default value is `1.`

Returns MCMC chain (2D array), array of rejected points (2D array), acceptance rate (float), and first lag autocorrelation of the thinned MCMC chain (float)

Return type tuple containing (ndarray, ndarray, float, float)

CHAPTER 10

Benchmarks

10.1 Rosenbrock Function Benchmark

This benchmark performs convergence tests on a single emulator with variable numbers of input parameters. The example is based on the Rosenbrock function (see https://www.sfu.ca/~ssurjano/rosen.html). This function can be defined in an artibrary number of dimensions, so it provides a useful test for how emulators based on increasing numbers of parameters perform as the size of the training data is varied. As the number of training points increases, the prediction error and prediction variance should decrease. However, this will depend on the number of dimensions in the function – in general, the size of the input space grows exponentially with the number of dimensions, while the samples drawn here grow linearly with the number of dimensions. Thus, the higher dimensional emulators will perform worse for the same number of samples per dimension.

10.2 Branin Function Benchmark

This benchmark performs convergence tests on multiple realizations of the 2D Branin function. Details of the 2D Branin function can be found at https://www.sfu.ca/~ssurjano/branin.html. This particular version uses 8 realizations of the Branin function, each with a different set of parameters. The code samples these 8 realizations simultaneously using a spacefilling Latin Hypercube experimental design with a varying number of target points, and then tests the convergence of the resulting emulators. As the number of targe points increases, the prediction error and prediction variance should decrease.

(Note however that eventually, the predictions worsen once the number of target points becomes large enough that the points become too densely sampled. In this case, the points become co-linear and the resulting covariance matrix is singular and cannot be inverted. To avoid this problem, the code iteratively adds additional noise to the covariance function to stabilize the inversion. However, this noise reduces the accuracy of the predictions. The values chosen for this benchmark attempt to avoid this, but in some cases this still becomes a problem due to the inherent smoothness of the squared exponential covariance function.)

10.3 Tsunami Data Benchmark

This benchmark examines the performance of emulators fit in parallel to multiple targets. The benchmark uses a set of tsunami simulations, where the inputs are 14 values of the seafloor displacement resulting from an earthquake, and the outputs are tsunami wave heights at different spatial locations. This benchmark fits 8, 16, 32, and 64 output points using 1, 2, 4, and 8 processes, and records the time required per emulator to perform the fitting. The actual performance will depend on the specific machine and the number of cores available. Once the number of processes exceeds the number of cores on the machine, the fitting time will increase, so the results will depend on the exact setup used. For reference, tests on a quad core MacBook Pro found that the fitting took roughly 1 second per emulator on a single core, with the time per emulator dropping by about a factor of 2 when 4 processes were used.

10.4 MCMC Benchmark

10.5 MICE Benchmark

This benchmark performs convergence tests using the MICE experimental design applied to the 2D Branin function. Details of the 2D Branin function can be found at https://www.sfu.ca/~ssurjano/branin.html. The code samples the Branin function using the MICE experimental design algorithm with a varying number of target points, and then tests the convergence of the resulting emulators. As the number of targe points increases, the prediction error and prediction variance should decrease.

(Note however that eventually, the predictions worsen once the number of target points becomes large enough that the points become too densely sampled. In this case, the points become co-linear and the resulting covariance matrix is singular and cannot be inverted. To avoid this problem, the code iteratively adds additional noise to the covariance function to stabilize the inversion. However, this noise reduces the accuracy of the predictions. The values chosen for this benchmark attempt to avoid this, but in some cases this still becomes a problem due to the inherent smoothness of the squared exponential covariance function.)

10.6 Dimension Reduction Benchmark

Dimension reduction benchmark (gKDR)

Plot L_1 model error against reduced dimension, for a one-hundred dimensional problem. The model is a Gaussian process (with maximum-likelihood hyperparameters), with inputs determined by gKDR, varying structural dimension and with a selection of scale parameters. The inputs to the model are 100 points chosen uniformly at random in 100 dimensions, and the observations are a linear mapping of these into one dimension.

10.7 History Matching Benchmark

Simple demos and sanity checks for the HistoryMatching class.

Provided methods:

get_y_simulated_1D: A toy model that acts as the simulator output for constructing GPEs for 1D data.

get_y_simulated_2D: A toy model that acts as the simulator output for constructing GPEs for 2D data.

demo_1D: Follows the example of http://www.int.washington.edu/talks/WorkShops/int_16_2a/People/Vernon_I/Vernon2.pdf in setting up a simple test model, constructing a gaussian process to emulate it, and ruling out expectations of the GPE based on an historical observation.

demo_2D: As demo_1D, however the toy model is expanded to take two inputs rather then 1 extending it to a second dimension. Exists primarily to confirm that the HistoryMatching class is functional with higher-dimensional parameter spaces.

The code includes a series of benchmarks that illustrate various pieces of the implementation. Benchmarks can be run from the `mogp_emulator/tests` directory by entering `make all` or `make benchmarks` to run all benchmarks, or `make rosenbrock`, `make branin`, `make tsunami`, `make mcmc`, `make mice`, `make gKDR`, or `make histmatch` to run the individual benchmarks.

10.8 Single Emulator Convergence Tests

The first benchmark examines the convergence of a single emulator applied to the Rosenbrock function in several different dimensions (more details can be found here). This illustrates how the emulator predictions improve as the number of training points is increased for different numbers of input parameters. The benchmark evaluates the Rosenbrock function in 4, 6, and 8 dimensions and shows that the mean squared prediction error and the mean variance improve with the number of training points used. Matplotlib can optionally be used to visualize the results.

10.9 Multi-Output Convergence Tests

The second benchmark examines the convergence of multiple emulators derived from the same input values. This benchmark is based on the 2D Branin function (more details on this function can be found here) . The code uses 8 different realizations of the Branin function using different parameter values, and then examines the convergence of the 8 different emulators fit using different number of parameter values based on the prediction errors and variance values. The results can optionally be visualized using Matplotlib.

10.10 Multi-Output Performance Benchmark

A performance benchmark is included that uses a set of Tsunami simulation results to examine the speed at which the code fits multiple emulators in parallel. The code fits 8, 16, 32, and 64 emulators using 1, 2, 4, and 8 processess and notes the time required to perform the fitting. Note that the results will depend on the number of cores on the computer – once you exceed the number of cores, the performance will degrade. As with the other benchmarks, Matplotlib can optionally be used to plot the results.

10.11 MCMC Benchmark

A benchmark applying the software to fitting an emulator with MCMC sampling is included. The code draws hyperparameter samples and compares the resulting posterior distributions with the values found via maximum likelihood estimation. If Matplotlib is installed, a histogram of the parameter samples is shown.

10.12 MICE Benchmark

A benchmark comparing the MICE Sequential design method to Latin Hypercube sampling is also available. This creates designs of a variety of sizes and computes the error on unseen data for the 2D Branin function. It compares the accuracy of the sequential design to the Latin Hypercube for both the predictions and uncertainties.

10.13 Dimension Reduction Benchmark

A benchmark illustrating gradient-based kernel dimension reduction is available in `tests/benchmark_kdr_GP.py` This problem contains a 100 dimension function with a single active dimension and shows how the loss depends on the number of dimensions included in the reduced dimension space.

10.14 History Matching Benchmark

A benchmark illustrating use of the `HistoryMatching` class to rule out parts of the input space using a GP emulator to make predictions. The benchmark contains a 1D and a 2D example.

CHAPTER 11

Indices and tables

- genindex
- modindex
- search

Bibliography

[Fukumizu1] https://www.ism.ac.jp/~fukumizu/software.html

[FL13] Fukumizu, Kenji and Chenlei Leng. “Gradient-based kernel dimension reduction for regression.” Journal of the American Statistical Association 109, no. 505 (2014): 359-370

Python Module Index

Index