Multi-Output GP Emulator Documentation

Release 0.1.0

Eric Daub

May 20, 2020

Contents:

CHAPTER 1

The `GaussianProcess` Class

Implementation of a Gaussian Process Emulator.

This class provides an interface to fit a Gaussian Process Emulator to a set of training data. The class can be initialized from either a pair of inputs/targets arrays, or a file holding data saved from a previous emulator instance (saved via the `save_emulator` method). Once the emulator has been created, the class provides methods for fitting optimal hyperparameters, changing hyperparameter values, making predictions, and other calculations associated with fitting and making predictions.

The internal emulator structure involves arrays for the inputs, targets, and hyperparameters. Other useful information are the number of training examples `n` and the number of input parameters `D`. These parameters are available externally through the `get_n` and `get_D` methods

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.get_n()
2
>>> gp.get_D()
3
>>> np.random.seed(47)
>>> gp.learn_hyperparameters()
(5.140462159403397, array([-13.02460687,  -4.02939647, -39.2203646 ,   3.25809653]))
>>> 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**(**args*)

Implementation of a Gaussian Process Emulator.

This class provides an interface to fit a Gaussian Process Emulator to a set of training data. The class can be initialized from either a pair of inputs/targets arrays, or a file holding data saved from a previous emulator instance (saved via the `save_emulator` method). Once the emulator has been created, the class provides methods for fitting optimal hyperparameters, changing hyperparameter values, making predictions, and other calculations associated with fitting and making predictions.

The internal emulator structure involves arrays for the inputs, targets, and hyperparameters. Other useful information are the number of training examples `n` and the number of input parameters `D`. These parameters are available externally through the `get_n` and `get_D` methods

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.get_n()
2
>>> gp.get_D()
3
>>> np.random.seed(47)
>>> gp.learn_hyperparameters()
(5.140462159403397, array([-13.02460687,  -4.02939647, -39.2203646 ,   3.
↪25809653]))
>>> 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__(**args*)

Create a new GP Emulator

Creates a new GP Emulator from either the input data and targets to be fit or a file holding the input/targets and (optionally) learned parameter values.

Arguments passed to the `__init__` method must be either two arguments which are numpy arrays `inputs` and `targets`, described below, three arguments which are the same `inputs` and `targets` arrays plus a float representing the `nugget` parameter, or a single argument which is the filename (string or file handle) of a previously saved emulator.

`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.

`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`.

`nugget` is the additional noise added to the emulator targets when fitting. This can take on values `None` (in which case, noise will be added adaptively to stabilize fitting), or a non-negative float (in which case, a fixed noise level will be used). If no value is specified for the `nugget` parameter, `None` is the default.

If two or three input arguments `inputs`, `targets`, and optionally `nugget` are given:

Parameters

- **inputs** (*ndarray*) – Numpy array holding emulator input parameters. Must be 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`
- **nugget** – Noise to be added to the diagonal or `None`. A float specifies the noise level explicitly, while if `None` is given, the noise will set to be as small as possible to ensure stable inversion of the covariance matrix. Optional, default is `None`.

If one input argument `emulator_file` is given:

Parameters **emulator_file** (*str or file*) – Filename or file object for saved emulator parameters (using the `save_emulator` method)

Returns New `GaussianProcess` instance

Return type *GaussianProcess*

get_D()
Returns number of inputs for the emulator

Returns Number of inputs for the emulator object

Return type int

get_n()
Returns number of training examples for the emulator

Returns Number of training examples for the emulator object

Return type int

get_nugget()
Returns emulator nugget parameter

Returns current value of the nugget parameter. If the nugget is selected adaptively, returns None.

Returns Current nugget value, either a float or `None`

Return type float or None

get_params()
Returns emulator parameters

Returns current parameters for the emulator as a numpy array if they have been fit. If no parameters have been fit, returns None.

Returns Current parameter values (numpy array of length `D + 1`), or `None` if the parameters have not been fit.

Return type ndarray or None

hessian(*theta*)
Calculate the Hessian of the negative log-likelihood

Calculate the Hessian of the negative log-likelihood 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 `hessian` method is called after evaluating the `loglikelihood` 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 `loglikelihood` and

partial_devs methods as well. If the function is evaluated with a different set of parameters than was previously used to set the log-likelihood, the method calls _set_params to compute the needed information. However, caling hessian does not evaluate the log-likelihood, so it does not change the cached values of the parameters or log-likelihood.

Parameters **theta** (*ndarray*) – Value of the hyperparameters. Must be array-like with shape (D + 1,)

Returns Hessian of the negative log-likelihood (array with shape (D + 1, D + 1))

Return type ndarray

learn_hyperparameters(*n_tries=15*, *theta0=None*, *method='L-BFGS-B'*, ***kwargs*)

Fit hyperparameters by attempting to minimize the negative log-likelihood

Fits the hyperparameters by attempting to minimize the negative log-likelihood 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).

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 the addition of additional noise added along the diagonal if adaptive noise was selected by setting the nugget parameter to be None), 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.

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 method returns the minimum negative log-likelihood found and the parameter values at which that minimum was obtained. The method also sets the current values of the hyperparameters to these optimal values and pre-computes the matrices needed to make predictions.

Parameters

- **n_tries** (*int*) – Number of attempts to minimize the negative log-likelihood 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 (D + 1,). 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 the minimization routine. see available parameters in scipy.optimize.minimize for details.

Returns Minimum negative log-likelihood values and hyperparameters (numpy array with shape (D + 1,)) used to obtain those values. The method also sets the current values of the hyperparameters to these optimal values and pre-computes the matrices needed to make predictions.

Return type tuple containing a float and an ndarray

loglikelihood(*theta*)

Calculate the negative log-likelihood at a particular value of the hyperparameters

Calculate the negative log-likelihood 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-likelihood and its derivatives.

In addition to returning the log-likelihood value, it stores the current value of the hyperparameters and log-likelihood in attributes of the object.

Parameters **theta** (*ndarray*) – Value of the hyperparameters. Must be array-like with shape `(D + 1,)`

Returns negative log-likelihood

Return type float

partial_devs(*theta*)

Calculate the partial derivatives of the negative log-likelihood

Calculate the partial derivatives of the negative log-likelihood 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 `partial_devs` method is called after evaluating the `loglikelihood` 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 `loglikelihood`, `partial_devs`, and `hessian` methods. If the function is evaluated with a different set of parameters than was previously used to set the log-likelihood, the method calls `_set_params` to compute the needed information. However, caling `partial_devs` does not evaluate the log-likelihood, so it does not change the cached values of the parameters or log-likelihood.

Parameters **theta** (*ndarray*) – Value of the hyperparameters. Must be array-like with shape `(D + 1,)`

Returns partial derivatives of the negative log-likelihood (array with shape `(D + 1,)`)

Return type ndarray

predict(*testing*, *do_deriv=True*, *do_unc=True*)

Make a prediction for a set of input vectors

Makes predictions for the emulator 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 is returned as an `(n_predict, )` shaped numpy array as the first return value from the method.

Optionally, the emulator can also calculate the uncertainties 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.

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)
- **do_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`.
- **do_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`.
- **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_predict,)` while the derivatives have shape `(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

save_emulator(*filename*)

Write emulators to disk

Method saves the emulator to disk using the given filename or file handle. The method writes the inputs and targets arrays to file. If the model has been assigned parameters, either manually or by fitting, those parameters are saved as well. Once saved, the emulator can be read by passing the file name or handle to the one-argument `__init__` method.

Parameters **filename** (*str or file*) – Name of file (or file handle) to which the emulator will be saved.

Returns None

set_nugget(*nugget*)

Set the nugget parameter for the emulator

Method for changing the `nugget` parameter for the emulator. When a new emulator is initilized, this is set to 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, if `nugget` is set to be `None`, the fitting routine will adaptively make the noise parameter as large as is needed to ensure that the emulator can be fit.

Parameters **nugget** (*None or float*) – Controls how noise is added to the emulator. If `nugget` is a nonnegative float, then this manually sets the noise parameter (if negative, this will lead to an error), with `nugget = 0` resulting in interpolation with no smoothing noise added. `nugget = None` will adaptively select the smallest value of the noise term that still leads to a stable inversion of the matrix. Default behavior is `nugget = None`.

Returns None

Return type None

CHAPTER 2

The `MultiOutputGP` Class

Implementation of a multiple-output Gaussian Process Emulator.

This class provides an interface to fit a Gaussian Process Emulator to multiple targets using the same input data. The class creates all of the necessary sub-emulators from the input data and provides interfaces to the `learn_hyperparameters` and `predict` methods of the sub-emulators. Because the emulators are all fit independently, the class provides the option to use multiple processes to fit the emulators and make predictions in parallel.

The emulators are stored internally in a list. Other useful information stored is the numer of emulators `n_emulators`, number of training examples `n`, and number of input parameters `D`. These other variables are made available externally through the `get_n_emulators`, `get_n`, and `get_D` methods.

Example:

```
>>> import numpy as np
>>> from mogp_emulator import MultiOutputGP
>>> x = np.array([[1., 2., 3.], [4., 5., 6.]])
>>> y = np.array([[4., 6.], [5., 7.]])
>>> mogp = MultiOutputGP(x, y)
>>> print(mogp)
Multi-Output Gaussian Process with:
2 emulators
2 training examples
3 input variables
>>> mogp.get_n_emulators()
2
>>> mogp.get_n()
2
>>> mogp.get_D()
3
>>> np.random.seed(47)
>>> mogp.learn_hyperparameters()
[(5.140462159403397, array([-13.02460687,  -4.02939647, -39.2203646 ,   3.25809653])),
 (5.322783716197557, array([-18.448741  ,  -5.46557813,  -4.81355357,   3.61091708]))]
>>> x_predict = np.array([[2., 3., 4.], [7., 8., 9.]])
```

(continues on next page)

(continued from previous page)

```
>>> mogp.predict(x_predict)
(array([[4.74687618, 6.84934016],
       [5.7350324 , 8.07267051]]),
 array([[0.01639298, 1.05374973],
       [0.01125792, 0.77568672]]),
 array([[[8.91363045e-05, 7.18827798e-01, 3.74439445e-16],
         [4.64005897e-06, 3.74191346e-02, 1.94917337e-17]],
        [[5.58461022e-07, 2.42945502e-01, 4.66315152e-01],
         [1.24593861e-07, 5.42016666e-02, 1.04035918e-01]]]))
```

class mogp_emulator.MultiOutputGP.**MultiOutputGP**(**args*)

Implementation of a multiple-output Gaussian Process Emulator.

This class provides an interface to fit a Gaussian Process Emulator to multiple targets using the same input data. The class creates all of the necessary sub-emulators from the input data and provides interfaces to the `learn_hyperparameters` and `predict` methods of the sub-emulators. Because the emulators are all fit independently, the class provides the option to use multiple processes to fit the emulators and make predictions in parallel.

The emulators are stored internally in a list. Other useful information stored is the numer of emulators `n_emulators`, number of training examples `n`, and number of input parameters `D`. These other variables are made available externally through the `get_n_emulators`, `get_n`, and `get_D` methods.

Example:

```
>>> import numpy as np
>>> from mogp_emulator import MultiOutputGP
>>> x = np.array([[1., 2., 3.], [4., 5., 6.]])
>>> y = np.array([[4., 6.], [5., 7.]])
>>> mogp = MultiOutputGP(x, y)
>>> print(mogp)
Multi-Output Gaussian Process with:
2 emulators
2 training examples
3 input variables
>>> mogp.get_n_emulators()
2
>>> mogp.get_n()
2
>>> mogp.get_D()
3
>>> np.random.seed(47)
>>> mogp.learn_hyperparameters()
[(5.140462159403397, array([-13.02460687,  -4.02939647, -39.2203646 ,   3.
↪25809653])),
 (5.322783716197557, array([-18.448741  ,  -5.46557813,  -4.81355357,   3.
↪61091708]))]
>>> x_predict = np.array([[2., 3., 4.], [7., 8., 9.]])
>>> mogp.predict(x_predict)
(array([[4.74687618, 6.84934016],
       [5.7350324 , 8.07267051]]),
 array([[0.01639298, 1.05374973],
       [0.01125792, 0.77568672]]),
 array([[[8.91363045e-05, 7.18827798e-01, 3.74439445e-16],
         [4.64005897e-06, 3.74191346e-02, 1.94917337e-17]],
        [[5.58461022e-07, 2.42945502e-01, 4.66315152e-01],
         [1.24593861e-07, 5.42016666e-02, 1.04035918e-01]]]))
```

__init__(**args*)

Create a new multi-output GP Emulator

Creates a new multi-output GP Emulator from either the input data and targets to be fit or a file holding the input/targets and (optionally) learned parameter values.

Arguments passed to the `__init__` method must be two or three arguments which are numpy arrays `inputs` and `targets` and optionally `nugget`, described below, or a single argument which is the filename (string or file handle) of a previously saved emulator.

`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. Because the model assumes all outputs are drawn from the same identical set of simulations (i.e. the normal use case is to fit a series of computer simulations with multiple outputs from the same input), the input to each emulator is identical.

`targets` is the target data to be fit by the emulator, also held in an array-like object. This can be either a 1D or 2D array, where the last dimension must have length `n`. If the `targets` array is of shape `(n_emulators,n)`, then the emulator fits a total of `n_emulators` to the different target arrays, while if targets has shape `(n,)`, a single emulator is fit.

`nugget` is a list or other iterable of nugget parameters for each emulator. Its length must match the number of targets to be fit. The values must be `None` (adaptive noise addition) or a non-negative float, and the emulators can have different noise behaviors.

If two or three input arguments `inputs`, `targets`, and optionally `nugget` are given:

Parameters

- **inputs** (*ndarray*) – Numpy array holding emulator input parameters. Must be 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 2D or 1D with length `n` in the final dimension. The first dimension is of length `n_emulators` (defaults to a single emulator if the input is 1D)
- **nugget** – `None` or list or other iterable holding values for nugget parameter for each emulator. Length must be `n_emulators`. Individual values can be `None` (adaptive noise addition), or a non-negative float. This parameter is optional, and defaults to `None`

If one input argument `emulator_file` is given:

Parameters **emulator_file** (*str or file*) – Filename or file object for saved emulator parameters (using the `save_emulator` method)

Returns New `MultiOutputGP` instance

Return type *MultiOutputGP*

get_D()

Returns number of inputs for each emulator

Returns Number of inputs for each emulator in the object

Return type int

get_n()

Returns number of training examples in each emulator

Returns Number of training examples in each emulator in the object

Return type int

get_n_emulators()
Returns the number of emulators

Returns Number of emulators in the object

Return type int

get_nugget()
Returns value of nugget for all emulators

Returns value of nugget for all emulators as a list. Values can be `None`, or a nonnegative float for each emulator.

Returns nugget values for all emulators (list of length `n_emulators` containint floats or `None`. nugget type and values can vary across all emulators if desired.)

Return type list

learn_hyperparameters(*n_tries=15*, *theta0=None*, *processes=None*, *method='L-BFGS-B'*, ***kwargs*)
Fit hyperparameters for each model

Fit the hyperparameters for each emulator. Options that can be specified include the number of different initial conditions to try during the optimization step, the level of verbosity of output during the fitting, the initial values of the hyperparameters to use when starting the optimization step, and the number of processes to use when fitting the models. Since each model can be fit independently of the others, parallelization can significantly improve the speed at which the models are fit.

Returns a list holding `n_emulators` tuples, each of which contains the minimum negative log-likelihood and a numpy array holding the optimal parameters found for each model.

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 the addition of additional "nugget" or noise added along the diagonal), the iteration is skipped. If all attempts to find optimal hyperparameters result in an error, then the method raises an exception.

Parameters

- **n_tries** (*int*) – (optional) The number of different initial conditions to try when optimizing over the hyperparameters (must be a positive integer, default = 15)
- **theta0** (*ndarray or None*) – (optional) Initial value of the hyperparameters to use in the optimization routine (must be array-like with a length of `D + 1`, where `D` is the number of input parameters to each model). Default is `None`.
- **processes** (*int or None*) – (optional) Number of processes to use when fitting the model. Must be a positive integer or `None` to use the number of processors on the computer (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 the minimization routine. see available parameters in `scipy.optimize.minimize` for details.

Returns List holding `n_emulators` tuples of length 2. Each tuple contains the minimum negative log-likelihood for that particular emulator and a numpy array of length `D + 2` holding the corresponding hyperparameters

Return type list

predict (*testing*, *do_deriv=True*, *do_unc=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 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.

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)
- **do_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`.
- **do_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`.
- **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

save_emulators (*filename*)
Write emulators to disk

Method saves emulators to disk using the given filename or file handle. The (common) inputs to all emulators are saved, and all targets are collected into a single numpy array (this saves the data in the same format used in the two-argument `__init__` method). If the model has been assigned parameters, either manually or by fitting, those parameters are saved as well. Once saved, the emulator can be read by passing the file name or handle to the one-argument `__init__` method.

Parameters **filename** (*str or file*) – Name of file (or file handle) to which the emulators will be saved.

Returns None

set_nugget (*nugget*)
Sets value of nugget for all emulators

Sets value of nugget for all emulators from values provided as a list or other iterable. Values can be `None`, or a nonnegative float for each emulator. The length of the input list must have length `n_emulators`.

Parameters **nugget** – List of nugget values for all emulators (must be of length `n_emulators` and contain floats or `None`. Nugget type and values can vary across all emulators if desired.)

CHAPTER 3

The `Kernel` Class

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_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

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

CHAPTER 4

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 gen-

erates 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

CHAPTER 5

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

CHAPTER 6

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

CHAPTER 7

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

CHAPTER 8

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=None*, *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=None*, *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 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_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

CHAPTER 9

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**(**args*)

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

CHAPTER 10

GP Emulator 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.

CHAPTER 11

Indices and tables

- genindex
- modindex
- search

Python Module Index

m

Index

Symbols

C

E

F

G

H

K

L

M

P

R

S