Multi-Fidelity Functions Documentation

Sander van Rijn

Jun 08, 2022

Contents

This is the documentation for the `mf2` package. For a short introduction with examples, have a look at the *Getting Started* page. Otherwise, you can look at the available functions in the package by category.

The `mf2` package provides consistent, efficient and tested Python implementations of a variety of multi-fidelity benchmark functions. The goal is to simplify life for numerical optimization researchers by saving time otherwise spent reimplementing and debugging the same common functions, and enabling direct comparisons with other work using the same definitions, improving reproducibility in general.

A multi-fidelity function usually reprensents an objective which should be optimized. The term 'multi-fidelity' refers to the fact, that multiple versions of the objective function exist which differ in the accuray to describe the real objective. A typical real-world example would be the aerodynamic efficiency of an airfoil, e.g., its drag value for a given lift value. The different fidelity levels are given by the accuracy of the evaluation method used to estimate the efficiency. Lower-fidelity versions of the objective function refer to less accurate, but simpler approximations of the objective, such as computational fluid dynamic simulations on rather coarse meshes, whereas higher fidelity levels refer to more accurate but also much more demaning evaluations such as prototype tests in wind tunnels. The hope of multi-fildelity optimization approaches is that many of the not-so-accurate but simple low-fidelity evaluations can be used to achieve improved results on the realistic high-fidelity version of the objective where only very few evaluations can be performed.

The only dependency of the mf2 package is the numpy package.

The source for this package is hosted at github.com/sjvrijn/mf2.

Last updated: (Jun 08, 2022)

CHAPTER 1

Contents

1.1 Installation

The recommended way to install *mf2* is with Python's *pip*:

```
python3 -m pip install --user mf2
```

or alternatively using *conda*:

```
conda install -c conda-forge mf2
```

For the latest version, you can install directly from source:

```
python3 -m pip install --user https://github.com/sjvrijn/mf2/archive/master.zip
```

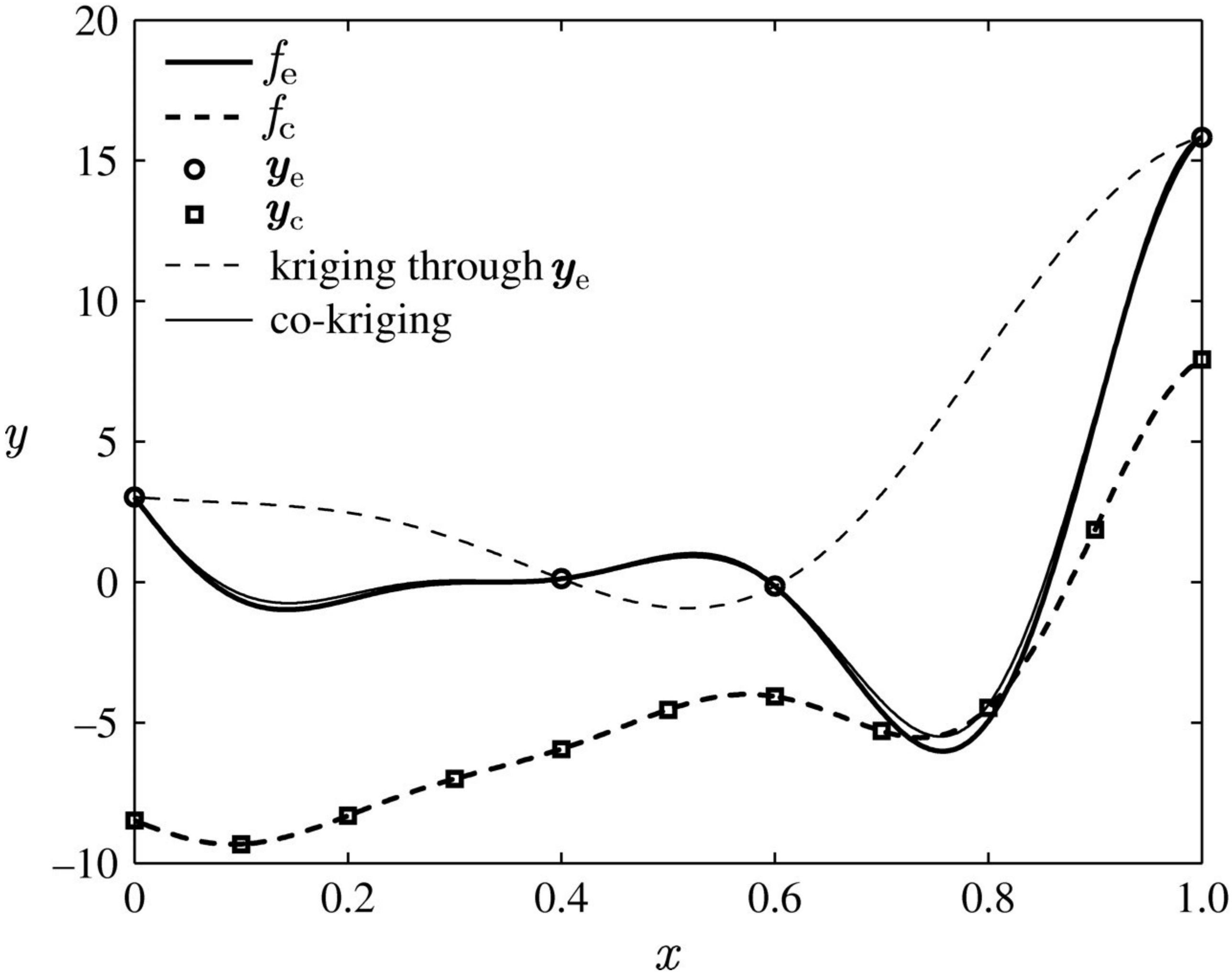
To work in your own version locally, it is best to clone the repository first:

```
git clone https://github.com/sjvrijn/mf2.git
cd mf2
python3 -m pip install --user -e .[dev]
```

1.2 Example Usage

This example is a reproduction of Figure 1 from http://doi.org/10.1098/rspa.2007.1900 :

The original figure:



Code to reproduce the above figure as close as possible:

```
# Typical imports: Matplotlib, numpy, sklearn and of course our mf2 package
import matplotlib.pyplot as plt
import mf2
import numpy as np
from sklearn.gaussian_process import GaussianProcessRegressor as GPR
from sklearn.gaussian_process import kernels

# Setting up
low_x = np.linspace(0, 1, 11).reshape(-1, 1)
high_x = low_x[[0, 4, 6, 10]]
diff_x = high_x

low_y = mf2.forrester.low(low_x)
high_y = mf2.forrester.high(high_x)
scale = 1.87  # As reported in the paper
diff_y = np.array([(mf2.forrester.high(x) - scale * mf2.forrester.low(x))[0]
                   for x in diff_x])

# Training GP models
kernel = kernels.ConstantKernel(constant_value=1.0) \
```

(continues on next page)

(continued from previous page)

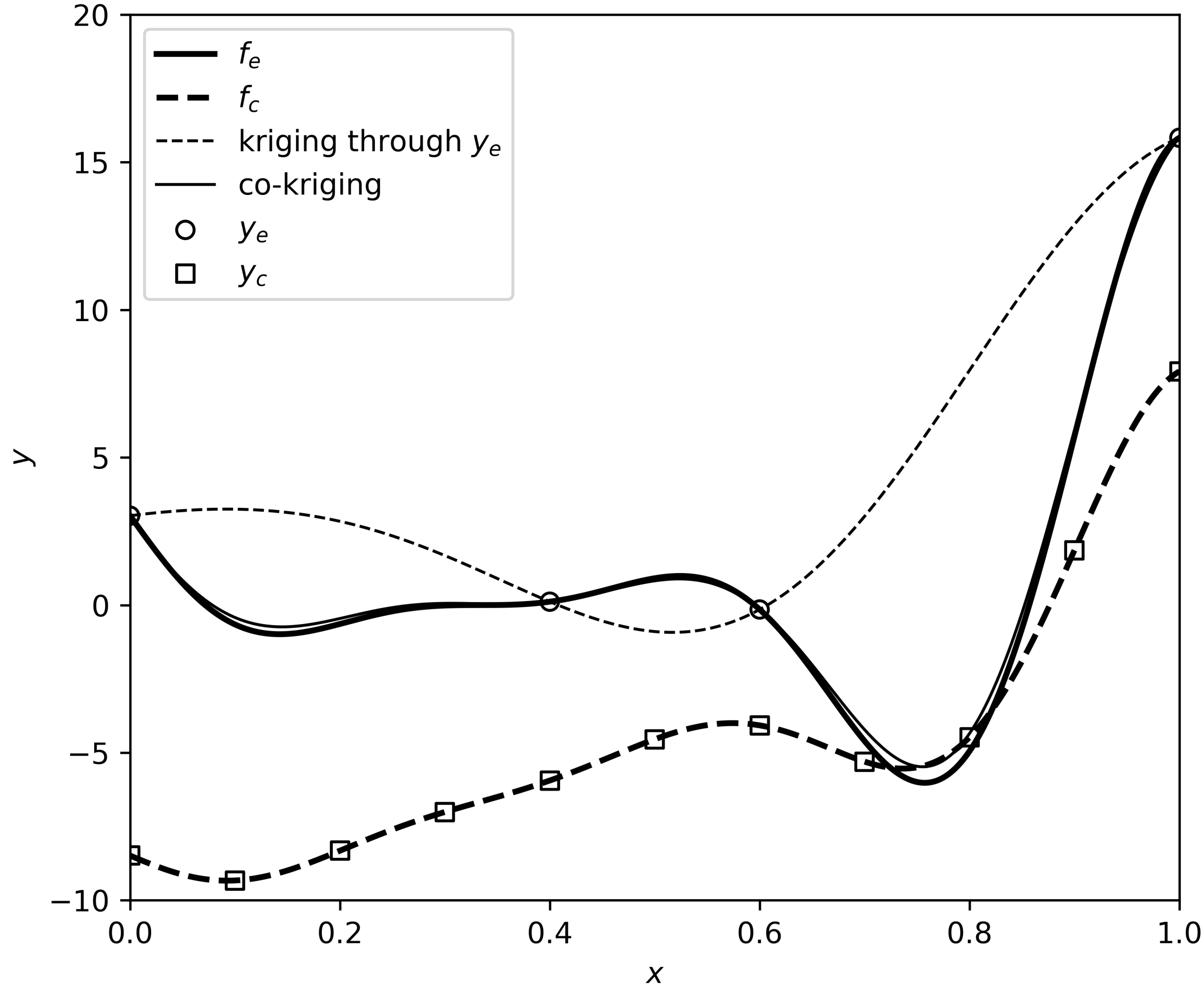
```
         * kernels.RBF(length_scale=1.0, length_scale_bounds=(1e-1, 10.0))

gp_direct = GPR(kernel=kernel).fit(high_x, high_y)
gp_low = GPR(kernel=kernel).fit(low_x, low_y)
gp_diff = GPR(kernel=kernel).fit(diff_x, diff_y)

# Using a simple function to combine the two models
def co_y(x):
    return scale * gp_low.predict(x) + gp_diff.predict(x)

# And finally recreating the plot
plot_x = np.linspace(start=0, stop=1, num=501).reshape(-1, 1)
plt.figure(figsize=(6, 5), dpi=600)
plt.plot(plot_x, mf2.forrester.high(plot_x), linewidth=2, color='black', label='$f_e$')
plt.plot(plot_x, mf2.forrester.low(plot_x), linewidth=2, color='black', linestyle='--',
         label='$f_c$')
plt.scatter(high_x, high_y, marker='o', facecolors='none', color='black', label='$y_e$')
plt.scatter(low_x, low_y, marker='s', facecolors='none', color='black', label='$y_c$')
plt.plot(plot_x, gp_direct.predict(plot_x), linewidth=1, color='black', linestyle='--',
         label='kriging through $y_e$')
plt.plot(plot_x, co_y(plot_x), linewidth=1, color='black', label='co-kriging')
plt.xlim([0, 1])
plt.ylim([-10, 20])
plt.xlabel('$x$')
plt.ylabel('$y$')
plt.legend(loc=2)
plt.tight_layout()
plt.savefig('../_static/recreating-forrester-2007.png')
plt.show()
```

Reproduced figure:



1.3 Performance

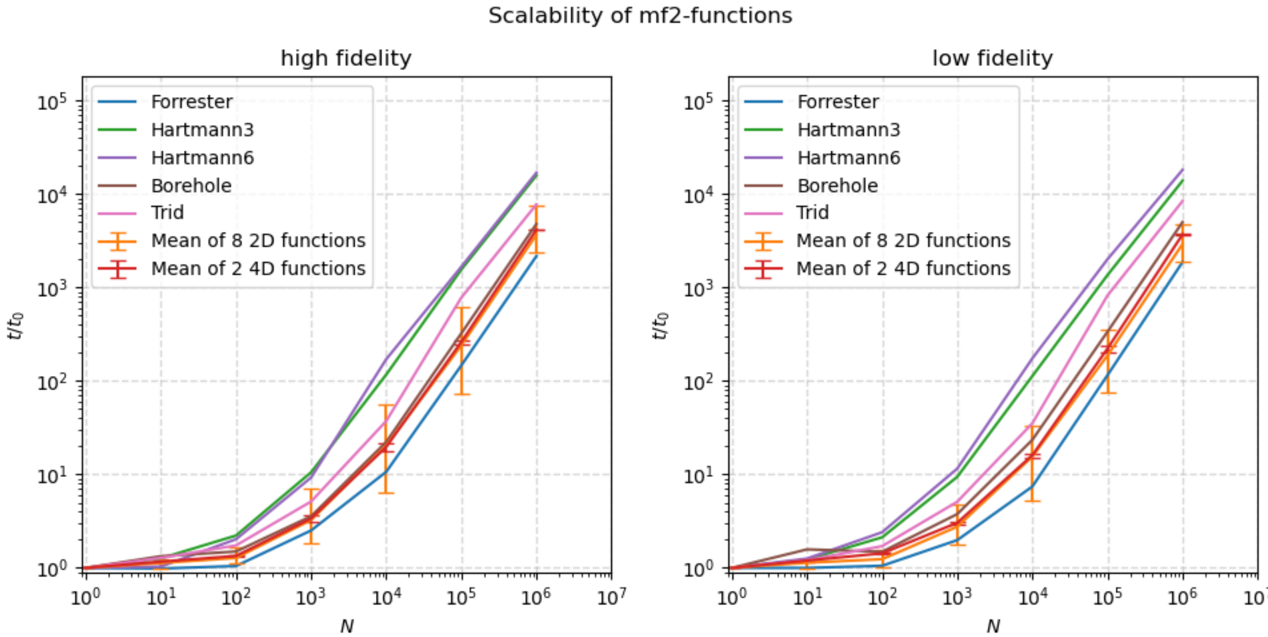
Where possible, all functions are written using numpy to make use of optimized routines and vectorization. Evaluating a single point typically takes less than 0.0001 seconds on a modern desktop system, regardless of function. This page shows some more detailed information about the performance, even though this library should not be a bottleneck in any programs.

The scripts for generating following performance overviews can be found in the docs/scripts folder of the repository. Presented running times were measured on a desktop PC with an Intel Core i7 5820k 6-core CPU, with Python 3.6.3 and Numpy 1.18.4.

1.3.1 Performance Scaling

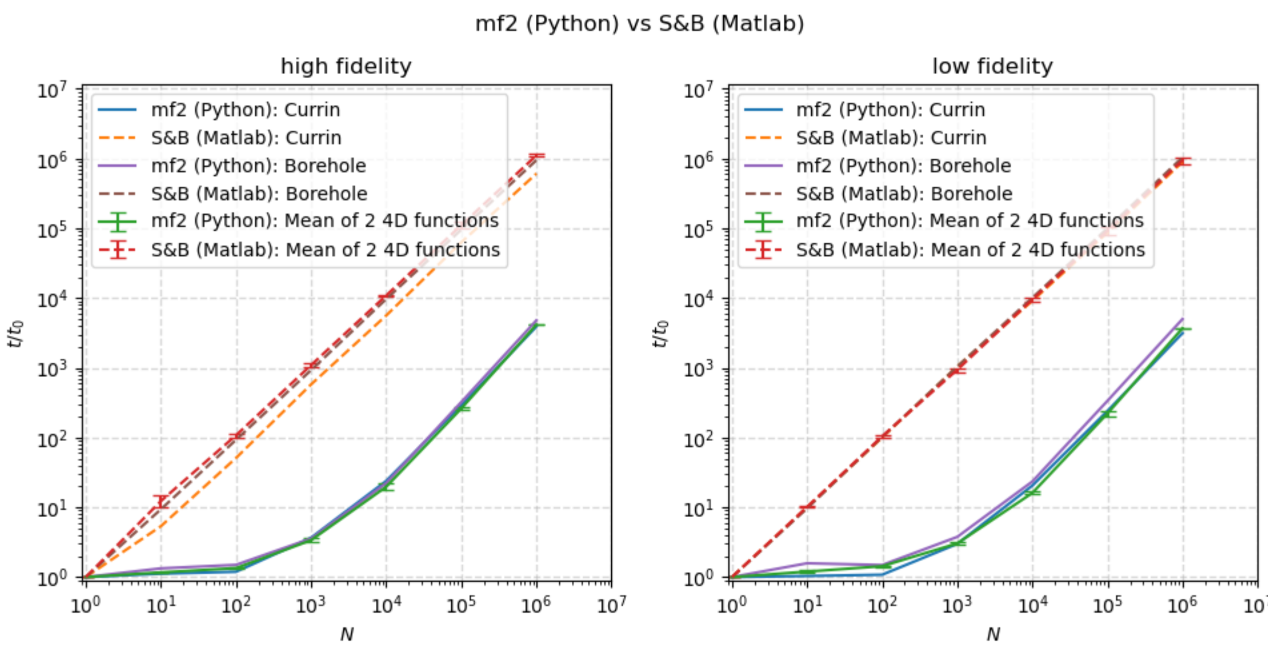
The image below shows how the runtime scales as `N` points are passed to the functions simultaneously as a matrix of size `(N, ndim)`. Performance for the high- and low-fidelity formulations are shown separately to give a fair comparison: many low-fidelities are defined as computations on top of the high-fidelity definitions. As absolute performance will vary per system, the runtime is divided by the time needed for `N=1` as a normalization. This is done independently for each function and fidelity level.

Up to `N=1_000`, the time required scales less than linearly thanks to efficient and vectorized numpy routines.



1.3.2 Performance Comparison

The following image shows how the scaling for the `mf2` implementation of the **Currin**, **Park91A**, **Park91B** and **Borehole** functions compares to the *Matlab* implementations by Surjanovic and Bingham, which can only evaluate one point at a time, so do not use any vectorization. Measurements were performed using *Matlab* version R2020a (9.8.0.1323502).



1.4 Getting Started

This page contains some explained examples to help get you started with using the `mf2` package.

1.4.1 The Basics: What's in a MultiFidelityFunction?

This package serves as a collection of functions with multiple fidelity levels. The number of levels is at least two, but differs by function. Each function is encoded as a *MultiFidelityFunction* with the following attributes:

.name The *name* is simply a standardized format of the name as an attribute to help identify which function is being represented[1] .

.ndim Number of dimensions. This is the dimensionality (i.e. length) of the input vector X of which the objective is evaluated.

.fidelity_names This is a list of the human-readable names given to each fidelity.

.u_bound, .l_bound The upper and lower bounds of the search-space for the function.

.functions A list of the actual function references. You won't typically need this list though, as will be explained next in *Accessing the functions*.

1.4.2 Simple Usage

Accessing the functions

As an example, we'll use the *booth* function. As we can see using `.ndim` and the bounds, it is two-dimensional:

```
>>> from mf2 import booth
>>> print(booth.ndim)
2
>>> print(booth.l_bound, booth.u_bound)
[-10. -10.] [10. 10.]
```

Most multi-fidelity functions in `mf2` are *bi-fidelity* functions, but a function can have any number of fidelities. A bi-fidelity function has two fidelity levels, which are typically called `high` and `low`. You can easily check the names of the fidelities by printing the `fidelity_names` attribute of a function:

```
>>> print(len(booth.fidelity_names))
2
>>> print(booth.fidelity_names)
['high', 'low']
```

These are just the names of the fidelities. The functions they represent can be accessed as an object-style *attribute*,

```
>>> print(booth.high)
<function booth_hf at 0x...>
```

as a dictionary-style *key*,

```
>>> print(booth['low'])
<function booth_lf at 0x...>
```

or with a list-style *index* (which just passes through to `.functions`).

[1] This is as they're instances of MultiFidelityFunction instead of separate classes.

```
>>> print(booth[0])
<function booth_hf at 0x...>
>>> print(booth[0] is booth.functions[0])
True
```

The object-style notation `function.fidelity()` is recommended for explicit access, but the other notations are available for more dynamic usage. With the list-style access, the *highest* fidelity is always at index *0*.

Calling the functions

All functions in the `mf2` package assume *row-vectors* as input. To evaluate the function at a single point, it can be given as a simple Python list or 1D numpy array. Multiple points can be passed to the function individually, or combined into a 2D list/array. The output of the function will always be returned as a 1D numpy array:

```
>>> X1 = [0.0, 0.0]
>>> print(booth.high(X1))
[74.]
>>> X2 = [
...     [ 1.0,  1.0],
...     [ 1.0, -1.0],
...     [-1.0,  1.0],
...     [-1.0, -1.0]
... ]
>>> print(booth.high(X2))
[ 20.  80.  72. 164.]
```

Using the bounds

Each function also has a given upper and lower bound, stored as a 1D numpy array. They will be of the same length, and exactly as long as the dimensionality of the function[2] .

Below is an example function to create a uniform sample within the bounds:

```
import numpy as np

def sample_in_bounds(func, n_samples):
    raw_sample = np.random.random((n_samples, func.ndim))

    scale = func.u_bound - func.l_bound
    sample = (raw_sample * scale) + func.l_bound

    return sample
```

1.4.3 Kinds of functions

Fixed Functions

The majority of multi-fidelity functions in this package are 'fixed' functions. This means that everything about the function is fixed:

- dimensionality of the input

[2] In fact, `.ndim` is defined as `len(self.u_bound)`

- number of fidelity levels
- relation between the different fidelity levels

Examples of these functions include the 2D *booth* and 8D *borehole* functions.

Dynamic Dimensionality Functions

Some functions are dynamic in the dimensionality of the input they accept. An example of such a function is the `forrester` function. The regular 1D function is included as `mf2.forrester`, but a custom n-dimensional version can be obtained by calling the factory:

```
forrester_4d = mf2.Forrester(ndim=4)
```

This `forrester_4d` is then a regular fixed function as seen before.

Adjustable Functions

Other functions have a tunable parameter that can be used to adjust the correlation between the different high and low fidelity levels. For these too, you can simply call a factory that will return a version of that function with the parameter fixed to your specification:

```
paciorek_high_corr = mf2.adjustable.paciorek(a2=0.1)
```

The exact relationship between the input parameter and resulting correlation can be found in the documentation of the specific functions. See for example *paciorek*.

1.4.4 Adding Your Own

Each function is stored as a `MultiFidelityFunction`-object, which contains the dimensionality, intended upper/lower bounds, and of course all fidelity levels. This class can also be used to define your own multi-fidelity function.

To do so, first define regular functions for each fidelity. Then create the `MultiFidelityFunction` object by passing a name, the upper and lower bounds, and a tuple of the functions for the fidelities.

The following is an example for a 1-dimensional multi-fidelity function named `my_mf_sphere` with three fidelities:

```
import numpy as np
from mf2 import MultiFidelityFunction

def sphere_hf(x):
    return x*x

def sphere_mf(x):
    return x * np.sqrt(x) * np.sign(x)

def sphere_lf(x):
    return np.abs(x)

my_mf_sphere = MultiFidelityFunction(
    name='sphere',
    u_bound=[1],
    l_bound=[-1],
    functions=(sphere_hf, sphere_mf, sphere_lf),
)
```

These functions can be accessed using list-style *indices*, but as no names are given, the object-style *attributes* or dict-style *keys* won't work:

```
>>> print(my_mf_sphere[0])
<function sphere_hf at 0x...>
>>> print(my_mf_sphere['medium'])
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
...
IndexError: Invalid index 'medium'
>>> print(my_mf_sphere.low)
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
...
AttributeError: 'MultiFidelityFunction' object has no attribute 'low'
>>> print(my_mf_sphere.fidelity_names)
None
```

To enable access by attribute or key, a tuple containing a name for each fidelity is required. Let's extend the previous example by adding `fidelity_names=('high', 'medium', 'low')`:

```
my_named_mf_sphere = MultiFidelityFunction(
    name='sphere',
    u_bound=[1],
    l_bound=[-1],
    functions=(sphere_hf, sphere_mf, sphere_lf),
    fidelity_names=('high', 'medium', 'low'),
)
```

Now we the attribute and key access will work:

```
>>> print(my_named_mf_sphere[0])
<function sphere_hf at 0x...>
>>> print(my_named_mf_sphere['medium'])
<function sphere_mf at 0x...>
>>> print(my_named_mf_sphere.low)
<function sphere_lf at 0x...>
>>> print(my_named_mf_sphere.fidelity_names)
('high', 'medium', 'low')
```

1.5 mf2 package

1.5.1 Fixed Functions

MultiFidelityFunction

multi_fidelity_function.py:

Defines the MultiFidelityFunction class for encapsuling all fidelities and parameters of a multi-fidelity function. Also contains any other utility functions that are commonly used by the various mf-functions in this package.

class AdjustableMultiFidelityFunction(*name*, *u_bound*, *l_bound*, *static_functions*, *adjustable_functions*, *fidelity_names=None*, *, *x_opt=None*)

Bases: `mf2.multi_fidelity_function.MultiFidelityFunction`

`__init__`(*name*, *u_bound*, *l_bound*, *static_functions*, *adjustable_functions*, *fidelity_names=None*, *, *x_opt=None*)

All fidelity levels and parameters of a multi-fidelity function.

Parameters

- **`name`** – Name of the multi-fidelity function.
- **`u_bound`** – Upper bound of the intended input range. Length is also used to determine the (fixed) dimensionality of the function.
- **`l_bound`** – Lower bound of the intended input range. Must be of same length as *u_bound*.
- **`static_functions`** – Iterable of function handles for the static, non-adjustable fidelities, sorted in *descending* order.
- **`adjustable_functions`** – Iterable of function handles for the adjustable fidelities, sorted in *descending* order.
- **`fidelity_names`** – List of names for the fidelities. Must be given to support dictionary- or attribute- style fidelity indexing, such as *f['high']()* and *f.high()*
- **`x_opt`** – Location of optimum x_opt for highest fidelity (if known).

`functions`

Combined static and adjustable functions

class MultiFidelityFunction(*name*, *u_bound*, *l_bound*, *functions*, *fidelity_names=None*, *, *x_opt=None*)

Bases: `object`

`__init__`(*name*, *u_bound*, *l_bound*, *functions*, *fidelity_names=None*, *, *x_opt=None*)

All fidelity levels and parameters of a multi-fidelity function.

Parameters

- **`name`** – Name of the multi-fidelity function.
- **`u_bound`** – Upper bound of the intended input range. Length is also used to determine the (fixed) dimensionality of the function.
- **`l_bound`** – Lower bound of the intended input range. Must be of same length as *u_bound*.
- **`functions`** – Iterable of function handles for the different fidelities, assumed to be sorted in *descending* order.
- **`fidelity_names`** – List of names for the fidelities. Must be given to support dictionary- or attribute-style fidelity indexing, such as *f['high']()* and *f.high()*
- **`x_opt`** – Location of optimum x_opt for highest fidelity (if known).

`bounds`

Lower and upper bounds as a single np.array of shape (2, ndim).

`functions`

`name`

`ndim`

Dimensionality of the function. Inferred as `len(self.u_bound)`.

`invert`(*mff: mf2.multi_fidelity_function.MultiFidelityFunction*) → mf2.multi_fidelity_function.MultiFidelityFunction

Invert a MultiFidelityFunction by multiplying all fidelities by -1

Parameters **`mff`** – The MultiFidelityFunction to invert

Returns A new MultiFidelityFunction with the inverted fidelities

Bohachevsky

Implementation of the bi-fidelity Bohachevsky function as defined in:

> Dong, H., Song, B., Wang, P. et al. Multi-fidelity information fusion based on prediction of kriging. Struct Multidisc Optim 51, 1267–1280 (2015) doi:10.1007/s00158-014-1213-9

Function definitions:

$$f_h(x_1, x_2) = x_1^2 + 2x_2^2 - 0.3\cos(3\pi x_1) - 0.4\cos(4\pi x_2) + 0.7$$

$$f_l(x_1, x_2) = f_h(0.7x_1, x_2) + x_1 x_2 - 12$$

bohachevsky = MultiFidelityFunction(Bohachevsky, [5. 5.], [-5. -5.], fidelity_names=['high
: 2D Bohachevsky function with fidelities 'high' and 'low'

bohachevsky_hf(*xx*)
: BOHACHEVSKY FUNCTION

  INPUT: xx = [x1, x2]

bohachevsky_lf(*xx*)
: BOHACHEVSKY FUNCTION, LOWER FIDELITY CODE Calls: bohachevsky_hf This function, from Dong et al. (2015), is used as the "low-accuracy code" version of the function bohachevsky_hf.

  INPUT: xx = [x1, x2]

l_bound = [-5, -5]
: Lower bound for Bohachevsky function

u_bound = [5, 5]
: Upper bound for Bohachevsky function

Booth

Implementation of the bi-fidelity Booth function as defined in:

> Dong, H., Song, B., Wang, P. et al. Multi-fidelity information fusion based on prediction of kriging. Struct Multidisc Optim 51, 1267–1280 (2015) doi:10.1007/s00158-014-1213-9

Function definitions:

$$f_h(x_1, x_2) = (x_1 + 2x_2 - 7)^2 + (2x_1 + x_2 - 5)^2$$

$$f_l(x_1, x_2) = f_h(0.4x_1, x_2) + 1.7x_1 x_2 - x_1 + 2x_2$$

booth = MultiFidelityFunction(Booth, [10. 10.], [-10. -10.], fidelity_names=['high', 'low']
: 2D Booth function with fidelities 'high' and 'low'

booth_hf(*xx*)
: BOOTH FUNCTION

  INPUT: xx = [x1, x2]

booth_lf(*xx*)
: BOOTH FUNCTION, LOWER FIDELITY CODE Calls: booth_hf This function, from Dong et al. (2015), is used as the "low-accuracy code" version of the function booth_hf.

  INPUT: xx = [x1, x2]

l_bound = [-10, -10]
Lower bound for Booth function

u_bound = [10, 10]
Upper bound for Booth function

Borehole

Implementation of the bi-fidelity Borehole function as defined in:

> Shifeng Xiong, Peter Z. G. Qian & C. F. Jeff Wu (2013) Sequential Design and Analysis of High-Accuracy and Low-Accuracy Computer Codes, Technometrics, 55:1, 37-46, DOI: 10.1080/00401706.2012.723572

Function definitions:

$$f_b(x, A, B) = \frac{A * T_u * (H_u - H_l)}{\left(\log(\frac{r}{r_w}) * (B + \frac{2L * T_u}{\log(\frac{r}{r_w}) * r_w^2 * K_w} + \frac{T_u}{T_l}\right)}$$

$$f_h(x) = f_b(x, 2\pi, 1)$$

$$f_l(x) = f_b(x, 5, 1.5)$$

Adapted from matlab implementation at

> https://www.sfu.ca/~ssurjano/borehole.html, retrieved 2017-10-02

by: Sonja Surjanovic and Derek Bingham, Simon Fraser University

Copyright 2013. Derek Bingham, Simon Fraser University.

THERE IS NO WARRANTY, EXPRESS OR IMPLIED. WE DO NOT ASSUME ANY LIABILITY FOR THE USE OF THIS SOFTWARE. If software is modified to produce derivative works, such modified software should be clearly marked. Additionally, this program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; version 2.0 of the License. Accordingly, this program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

borehole = MultiFidelityFunction(Borehole, [1.5000e-01 5.0000e+04 1.1560e+05 1.1100e+03 1.
8D Borehole function with fidelities 'high' and 'low'

borehole_hf(*xx*)
BOREHOLE FUNCTION

INPUT AND OUTPUT: inputs = [rw, r, Tu, Hu, Tl, Hl, L, Kw] output = water flow rate

borehole_lf(*xx*)
BOREHOLE FUNCTION, LOWER FIDELITY CODE This function is used as the "low-accuracy code" version of the function borehole_hf.

INPUT AND OUTPUT: inputs = [rw, r, Tu, Hu, Tl, Hl, L, Kw] output = water flow rate

l_bound = [0.05, 100, 63070, 990, 63.1, 700, 1120, 9855]
Lower bound for Borehole function

u_bound = [0.15, 50000, 115600, 1110, 116, 820, 1680, 12045]
Upper bound for Borehole function

Branin

Implementation of the bi-fidelity Branin function as defined in:

> Dong, H., Song, B., Wang, P. et al. Multi-fidelity information fusion based on prediction of kriging. Struct Multidisc Optim 51, 1267–1280 (2015) doi:10.1007/s00158-014-1213-9

Function definitions:

$$f_b(x_1, x_2) = \left(x_2 - (5.1\frac{x_1^2}{4\pi^2}) + \frac{5x_1}{\pi} - 6 \right)^2 + \left(10\cos(x_1)(1 - \frac{1}{8\pi} \right) + 10$$

$$f_h(x_1, x_2) = f_b(x_1, x_2) - 22.5x_2$$

$$f_l(x_1, x_2) = f_b(0.7x_1, 0.7x_2) - 15.75x_2 + 20(0.9 + x_1)^2 - 50$$

branin = MultiFidelityFunction(Branin, [10. 15.], [-5. 0.], fidelity_names=['high', 'low'])
: 2D Branin function with fidelities 'high' and 'low'

branin_base(*xx*)
: BRANIN FUNCTION

  INPUT: xx = [x1, x2]

branin_hf(*xx*)
: BRANIN FUNCTION, HIGH FIDELITY CODE Calls: branin_base This function, from Dong et al. (2015), is used as the "high-accuracy code" version of the function based on the 'traditional' branin function.

  INPUT: xx = [x1, x2]

branin_lf(*xx*)
: BRANIN FUNCTION, LOWER FIDELITY CODE Calls: branin_base This function, from Dong et al. (2015), is used as the "low-accuracy code" version of the function branin_hf.

  INPUT: xx = [x1, x2]

l_bound = [-5, 0]
: Lower bound for Branin function

u_bound = [10, 15]
: Upper bound for Branin function

Currin

Implementation of the bi-fidelity Currin function as defined in:

> Shifeng Xiong, Peter Z. G. Qian & C. F. Jeff Wu (2013) Sequential Design and Analysis of High-Accuracy and Low-Accuracy Computer Codes, Technometrics, 55:1, 37-46, DOI: 10.1080/00401706.2012.723572

Function definitions:

$$f_h(x_1, x_2) = \left(1 - \exp(-\frac{1}{2x_2}) \right) \frac{2300x_1^3 + 1900x_1^2 + 2092x_1 + 60}{100x_1^3 + 500x_1^2 + 4x_1 + 20}$$

$$\begin{aligned} f_l(x_1, x_2) = (&f_h(x_1 + 0.05, x_2 + 0.05) + \\ &f_h(x_1 + 0.05, x_2 - 0.05) + \\ &f_h(x_1 - 0.05, x_2 + 0.05) + \\ &f_h(x_1 - 0.05, x_2 - 0.05))/4 \end{aligned}$$

Adapted from matlab implementation at

https://www.sfu.ca/~ssurjano/curretal88exp.html, retrieved 2017-10-02

by: Sonja Surjanovic and Derek Bingham, Simon Fraser University

currin = MultiFidelityFunction(Currin, [1. 1.], [0. 0.], fidelity_names=['high', 'low'])
: 2D Currin function with fidelities 'high' and 'low'

currin_hf(*xx*)
: CURRIN ET AL. (1988) EXPONENTIAL FUNCTION

  INPUT: xx = [x1, x2]

currin_lf(*xx*)
: CURRIN ET AL. (1988) EXPONENTIAL FUNCTION, LOWER FIDELITY CODE Calls: currin_hf This function, from Xiong et al. (2013), is used as the "low-accuracy code" version of the function currin_hf.

  INPUT: xx = [x1, x2]

l_bound = [0, 0]
: Lower bound for Currin function

u_bound = [1, 1]
: Upper bound for Currin function

Forrester

forrester.py: Forrester function

This file contains the definition of an adapted version of the simple 1D example function as presented in:

> Forrester Alexander I.J, Sóbester András and Keane Andy J "Multi-fidelity Optimization via Surrogate Modelling", Proceedings of the Royal Society A, vol. 463, http://doi.org/10.1098/rspa.2007.1900

Function definitions:

$$f_h(x) = (6x-2)^2 \sin(12x-4)$$

$$f_l(x) = Af_h(x) + B(x-0.5) + C$$

With $A = 0.5, B = 10$ and $C = -5$ as recommended parameters.

This version has been adapted to be multi-dimensional, input can be arbitrarily many dimensions. Output value is calculated as the mean of the outcomes for all separate dimensions.

Forrester(*ndim: int*)
: Factory method for *ndim*-dimensional multi-fidelity Forrester function

  Parameters ndim – Desired dimensionality

  Returns `MultiFidelityFunction` instance with bounds of appropriate length

forrester = MultiFidelityFunction(Forrester, [1.], [0.], fidelity_names=['high', 'low'])
: 1D Forrester function with fidelities 'high' and 'low'

forrester_high (*xx*)

forrester_low (*xx*, ***, *A=0.5*, *B=10*, *C=-5*)

forrester_sf = MultiFidelityFunction(Forrester Single Fidelity, [1.], [0.], fidelity_names=
    1D Forrester function with single fidelity 'high'

l_bound = [0]
    Lower bound for Forrester function

u_bound = [1]
    Upper bound for Forrester function

Hartmann6

hartmann.py: contains the Hartmann6 function

As defined in

"Remarks on multi-fidelity surrogates" by Chanyoung Park, Raphael T. Haftka and Nam H. Kim (2016)

Function definitions:

$$f_h(x_1, ..., x_6) = -\frac{1}{1.94}\left(2.58 + \sum_{i=1}^{4} \alpha_i \exp\left(-\sum_{j=1}^{6} A_{ij}(x_j - P_{ij})^2\right)\right)$$

$$f_l(x_1, ..., x_6) = -\frac{1}{1.94}\left(2.58 + \sum_{i=1}^{4} \alpha'_i f_{exp}\left(-\sum_{j=1}^{6} A_{ij}(x_j - P_{ij})^2\right)\right)$$

$$f_{exp}(x) = (\exp(-4/9) + (\exp(-4/9) * (x+4)/9))^9$$

with the following matrices and vectors:

$$A = \begin{pmatrix} 10 & 3 & 17 & 3.5 & 1.7 & 8 \\ 0.05 & 10 & 17 & 0.1 & 8 & 14 \\ 3 & 3.5 & 1.70 & 10 & 17 & 8 \\ 17 & 8 & 0.05 & 10 & 0.1 & 14 \end{pmatrix}$$

$$P = 10^{-4}\begin{pmatrix} 1312 & 1696 & 5569 & 124 & 8283 & 5886 \\ 2329 & 4135 & 8307 & 3736 & 1004 & 9991 \\ 2348 & 1451 & 3522 & 2883 & 3047 & 6650 \\ 4047 & 8828 & 8732 & 5743 & 1091 & 381 \end{pmatrix}$$

$$\alpha = \{0.5, 0.5, 2.0, 4.0\}$$
$$\alpha' = \{1.0, 1.2, 3.0, 3.2\}$$

hartmann6 = MultiFidelityFunction(Hartmann6, [1. 1. 1. 1. 1. 1.], [0.1 0.1 0.1 0.1 0.1 0.1]
    6D Hartmann6 function with fidelities 'high' and 'low'

hartmann6_hf (*xx*)

hartmann6_lf (*xx*)

l_bound = [0.1, 0.1, 0.1, 0.1, 0.1, 0.1]
    Lower bound for Hartmann6 function

u_bound = [1, 1, 1, 1, 1, 1]
    Upper bound for Hartmann6 function

Himmelblau

Implementation of the bi-fidelity Himmelblau function as defined in:

> Dong, H., Song, B., Wang, P. et al. Multi-fidelity information fusion based on prediction of kriging. Struct Multidisc Optim 51, 1267–1280 (2015) doi:10.1007/s00158-014-1213-9

Function definitions:

$$f_h(x_1, x_2) = (x_1^2 + x_2 - 11)^2 + (x_2^2 + x_1 - 7)^2$$

$$f_l(x_1, x_2) = f_h(0.5x_1, 0.8x_2) + x_2^3 - (x_1 + 1)^2$$

himmelblau = MultiFidelityFunction(Himmelblau, [4. 4.], [-4. -4.], fidelity_names=['high',

2D Himmelblau function with fidelities 'high' and 'low'

himmelblau_hf (*xx*)

HIMMELBLAU FUNCTION

INPUT: xx = [x1, x2]

himmelblau_lf (*xx*)

HIMMELBLAU FUNCTION, LOWER FIDELITY CODE Calls: himmelblau_hf This function, from Dong et al. (2015), is used as the "low-accuracy code" version of the function himmelblau_hf.

INPUT: xx = [x1, x2]

l_bound = [-4, -4]

Lower bound for Himmelblau function

u_bound = [4, 4]

Upper bound for Himmelblau function

Park91 A

Implementation of the bi-fidelity Park ('91) A function as defined in:

> Shifeng Xiong, Peter Z. G. Qian & C. F. Jeff Wu (2013) Sequential Design and Analysis of High-Accuracy and Low-Accuracy Computer Codes, Technometrics, 55:1, 37-46, DOI: 10.1080/00401706.2012.723572

Function definitions:

$$f_h(x_1, x_2, x_3, x_4) = \frac{x_1}{2}\left(\sqrt{1 + (x_2 + x_3^2) * \frac{x_4}{x_1^2}} - 1\right) + (x_1 + 3x_4)\exp(1 + \sin(x_3))$$

$$f_l(x_1, x_2, x_3, x_4) = (1 + \sin(x_1)/10) f_h(x_1, x_2, x_3, x_4) + -2x_1 + x_2^2 + x_3^2 + 0.5$$

Adapted from matlab implementation at

> https://www.sfu.ca/~ssurjano/park91a.html, retrieved 2017-10-02

by: Sonja Surjanovic and Derek Bingham, Simon Fraser University

l_bound = [1e-08, 0, 0, 0]
Lower bound for Park91A function

park91a = MultiFidelityFunction(Park91A, [1. 1. 1. 1.], [1.e-08 0.e+00 0.e+00 0.e+00], fide
4D Park91A function with fidelities 'high' and 'low'

park91a_hf(*xx*)
PARK (1991) FUNCTION 1

INPUT: xx = [x1, x2, x3, x4]

park91a_lf(*xx*)
PARK (1991) FUNCTION 1, LOWER FIDELITY CODE Calls: park91a_hf This function, from Xiong et al. (2013), is used as the "low-accuracy code" version of the function park91a_hf.

INPUT: xx = [x1, x2, x3, x4]

u_bound = [1, 1, 1, 1]
Upper bound for Park91A function

Park91 B

Implementation of the bi-fidelity Park ('91) B function as defined in:

Shifeng Xiong, Peter Z. G. Qian & C. F. Jeff Wu (2013) Sequential Design and Analysis of High-Accuracy and Low-Accuracy Computer Codes, Technometrics, 55:1, 37-46, DOI: 10.1080/00401706.2012.723572

Function definitions:

$$f_h(x_1, x_2, x_3, x_4) = \frac{2}{3}\exp(x_1 + x_2) - x_4 \sin(x_3) + x_3$$

$$f_l(x_1, x_2, x_3, x_4) = 1.2 f_h(x_1, x_2, x_3, x_4) - 1$$

Adapted from matlab implementation at

https://www.sfu.ca/~ssurjano/park91b.html, retrieved 2017-10-02

by: Sonja Surjanovic and Derek Bingham, Simon Fraser University

l_bound = [0, 0, 0, 0]
Lower bound for Park91B function

park91b = MultiFidelityFunction(Park91B, [1. 1. 1. 1.], [0. 0. 0. 0.], fidelity_names=['hig
4D Park91B function with fidelities 'high' and 'low'

park91b_hf(*xx*)
PARK (1991) FUNCTION 2

INPUT: xx = [x1, x2, x3, x4]

park91b_lf(*xx*)
PARK (1991) FUNCTION 2, LOWER FIDELITY CODE Calls: park91b_hf This function, from Xiong et al. (2013), is used as the "low-accuracy code" version of the function park91b_hf.

INPUT: xx = [x1, x2, x3, x4]

u_bound = [1, 1, 1, 1]
Upper bound for Park91B function

Six-Hump Camelback

Implementation of the bi-fidelity Six-hump Camel-back function as defined in:

> Dong, H., Song, B., Wang, P. et al. Multi-fidelity information fusion based on prediction of kriging. Struct Multidisc Optim 51, 1267–1280 (2015) doi:10.1007/s00158-014-1213-9

Function definitions:

$$f_h(x_1, x_2) = 4x_1^2 - 2.1x_1^4 + \frac{x_1^6}{3} + x_1x_2 - 4x_2^2 + 4x_2^4$$

$$f_l(x_1, x_2) = f_h(0.7x_1, 0.7x_2) + x_1x_2 - 15$$

l_bound = [-2, -2]
Lower bound for Six-hump Camelback function

six_hump_camelback = MultiFidelityFunction(Six Hump Camelback, [2. 2.], [-2. -2.], fidelity
2D Six-hump Camelback function with fidelities 'high' and 'low'

six_hump_camelback_hf(*xx*)
SIX-HUMP CAMEL-BACK FUNCTION

INPUT: xx = [x1, x2]

six_hump_camelback_lf(*xx*)
SIX-HUMP CAMEL-BACK FUNCTION, LOWER FIDELITY CODE Calls: sixHumpCamelBack_hf This function, from Dong et al. (2015), is used as the "low-accuracy code" version of the function sixHumpCamelBack_hf.

INPUT: xx = [x1, x2]

u_bound = [2, 2]
upper bound for Six-hump Camelback function

1.5.2 Adjustable Functions

Adjustable Branin

Implementation of the adjustable bi-fidelity Branin function as defined in:

> Toal, D.J.J. Some considerations regarding the use of multi- fidelity Kriging in the construction of surrogate models. Struct Multidisc Optim 51, 1223–1245 (2015) doi:10.1007/s00158-014-1209-5

Function definitions:

$$f_h(x_1, x_2) = \left(x_2 - (5.1\frac{x_1^2}{4\pi^2}) + \frac{5x_1}{\pi} - 6\right)^2 + \left(10\cos(x_1)(1 - \frac{1}{8\pi}\right) + 10$$

$$f_l(x_1, x_2) = f_h(x_1, x_2) - (a + 0.5)\left(\left(x_2 - (5.1\frac{x_1^2}{4\pi^2}) + \frac{5x_1}{\pi} - 6\right)^2\right)$$

where $a \in [0, 1]$ is the adjustable parameter.

Note that f_h is equal to the non-adjustable f_b defined in *mf2.branin*.

adjustable_branin_lf (*xx*, *a*)

Adjustable Paciorek

Implementation of the adjustable bi-fidelity Paciorek function as defined in:

> Toal, D.J.J. Some considerations regarding the use of multi- fidelity Kriging in the construction of surrogate models. Struct Multidisc Optim 51, 1223–1245 (2015) doi:10.1007/s00158-014-1209-5

Function definitions:

$$f_h(x_1, x_2) = \sin\left(\frac{1}{x_1 x_2}\right)$$

$$f_l(x_1, x_2) = f_h(x_1, x_2) - 9a^2 \cos\left(\frac{1}{x_1 x_2}\right)$$

where $a \in (0, 1]$ is the adjustable parameter

adjustable_paciorek_lf (*xx*, *a*)

paciorek_hf (*xx*)

Adjustable Hartmann3

Implementation of the adjustable bi-fidelity Hartmann3 function as defined in:

> Toal, D.J.J. Some considerations regarding the use of multi- fidelity Kriging in the construction of surrogate models. Struct Multidisc Optim 51, 1223–1245 (2015) doi:10.1007/s00158-014-1209-5

Function definitions:

$$f_h(x_1, x_2, x_3) = -\sum_{i=1}^{4} \alpha_i \exp\left(-\sum_{j=1}^{3} \beta_{ij}(x_j - P_{ij})^2\right)$$

$$f_l(x_1, x_2, x_3) = -\sum_{i=1}^{4} \alpha_i \exp\left(-\sum_{j=1}^{3} \beta_{ij}\left(x_j - \frac{3}{4}P_{ij}(a+1)\right)^2\right)$$

with the following matrices and vectors:

$$\beta = \begin{pmatrix} 3 & 10 & 30 \\ 0.1 & 10 & 35 \\ 3 & 10 & 30 \\ 0.1 & 10 & 35 \end{pmatrix}$$

$$P = 10^{-4}\begin{pmatrix} 3689 & 1170 & 2673 \\ 4699 & 4387 & 7470 \\ 1091 & 8732 & 5547 \\ 381 & 5743 & 8828 \end{pmatrix}$$

$$\alpha = \{1.0, 1.2, 3.0, 3.2\}$$

and where $a \in [0, 1]$ is the adjustable parameter.

adjustable_hartmann3_lf (*xx*, *a*)

hartmann3_hf (*xx*)

Adjustable Trid

Implementation of the adjustable bi-fidelity Trid function as defined in:

> Toal, D.J.J. Some considerations regarding the use of multi- fidelity Kriging in the construction of surrogate models. Struct Multidisc Optim 51, 1223–1245 (2015) doi:10.1007/s00158-014-1209-5

$$f_h(x_1, ..., x_{10}) = \sum_{i=1}^{10}(x_i - 1)^2 - \sum_{i=2}^{10} x_i x_{i-1}$$

$$f_l(x_1, ..., x_{10}) = \sum_{i=1}^{10}(x_i - a)^2 - (a - 0.65) \sum_{i=2}^{10} x_i x_{i-1}$$

where $a \in [0, 1]$ is the adjustable parameter

adjustable_trid_lf(*xx*, *a*)

trid_hf(*xx*)

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

m

Index

Symbols

A

B

C

F

H

I

L

M

N

P

S

T

U