pySOT Documentation

Release

David Eriksson, David Bindel, Christine Shoemaker

Jul 21, 2017

User Documentation

This is the documentation for the Surrogate Optimization Toolbox (pySOT) for global deterministic optimization problems. pySOT is hosted on GitHub: https://github.com/dme65/pySOT.

The main purpose of the toolbox is for optimization of computationally expensive black-box objective functions with continuous and/or integer variables. We support inequality constraints of any form through a penalty method approach, but cannot yet efficiently handle equality constraints. All variables are assumed to have bound constraints in some form where none of the bounds are infinity. The tighter the bounds, the more efficient are the algorithms since it reduces the search region and increases the quality of the constructed surrogate. The longer the objective functions take to evaluate, the more efficient are these algorithms. For this reason, this toolbox may not be very efficient for problems with computationally cheap function evaluations. Surrogate models are intended to be used when function evaluations take from several minutes to several hours or more.

For easier understanding of the algorithms in this toolbox, it is recommended and helpful to read these papers. If you have any questions, or you encounter any bugs, please feel free to either submit a bug report on GitHub (recommended) or to contact me at the email address: dme65@cornell.edu. Keep an eye on the GitHub repository for updates and changes to both the toolbox and the documentation.

The toolbox is based on the following published papers: [*1*], [*2*], [*3*], [*4*], [*5*], [*6*].

CHAPTER 1

Quickstart

Dependencies

Before starting you will need Python 2.7.x or Python 3. You need to have numpy, scipy, and pip installed and we recommend installing Anaconda/Miniconda for your desired Python version.

There are a couple of optional components of pySOT that needs to be installed manually:

1. **scikit-learn**: Necessary in order to use the Gaussian process regression. The minimum version is 0.18.1. Can be installed using

   ```
   pip install "scikit-learn >= 0.18.1"
   ```

2. **py-earth**: Implementation of MARS. Can be installed using:

   ```
   pip install six http://github.com/scikit-learn-contrib/py-earth/tarball/master
   ```

   or

   ```
   git clone git://github.com/scikit-learn-contrib/py-earth.git
   cd py-earth
   pip install six
   python setup.py install
   ```

3. **mpi4py**: This module is necessary in order to use pySOT with MPI. Can be installed through pip:

   ```
   pip install mpi4py
   ```

   or through conda (Anaconda/Miniconda) where it can be channeled with your favorite MPI implementation such as mpich:

   ```
   conda install --channel mpi4py mpich mpi4py
   ```

4. **subprocess32**: A backport of the subprocess module for Python 3.2 that works for Python 2.7. This is the recommended way of launching workers through subprocesses for Python 2.7 and this module is easily installed using:

```
pip install subprocess32
```

5. **matlab_wrapper**: A module that can be used to create MATLAB sessions for older MATLAB versions where there is no default MATLAB engine. Easily instead using:

```
pip install matlab_wrapper
```

6. **PySide**: If you want to use the GUI you need to install PySide. This can be done with pip:

```
pip install PySide
```

Installation

There are currently two ways to install pySOT:

1. **(Recommended)** The easiest way to install pySOT is through pip in which case the following command should suffice:

```
pip install pySOT
```

2. The other option is cloning the repository and installing.

2.1. Clone the repository:

```
git clone https://github.com/dme65/pySOT
```

2.2. Navigate to the repository using:

```
cd pySOT
```

2.3. Install pySOT (you may need to use sudo for UNIX):

```
python setup.py install
```

Several examples problems are available at ./pySOT/test or in the pySOT.test module

CHAPTER 2

Surrogate optimization

Surrogate optimization algorithms generally consist of four components:

1. **Optimization problem:** All of the available information about the optimization problem, e.g., dimensionality, variable types, objective function, etc.
2. **Surrogate model:** Approximates the underlying objective function. Common choices are RBFs, Kriging, MARS, etc.
3. **Experimental design:** Generates an initial set of points for building the initial surrogate model
4. **Adaptive sampling:** Method for choosing evaluations after the experimental design has been evaluated.

The surrogate model (or response surfaces) is used to approximate an underlying function that has been evaluated for a set of points. During the optimization phase information from the surrogate model is used in order to guide the search for improved solutions, which has the advantage of not needing as many function evaluations to find a good solution. Most surrogate model algorithms consist of the same steps as shown in the algorithm below.

The general framework for a Surrogate Optimization algorithm is the following:

Inputs: Optimization problem, Experimental design, Adaptive sampling method, Surrogate model, Stopping criterion, Restart criterion

```
Generate an initial experimental design
Evaluate the points in the experimental design
Build a Surrogate model from the data
Repeat until stopping criterion met
   If restart criterion met
      Reset the Surrogate model and the adaptive sampling method
      go to 1
   Use the adaptive sampling method to generate new point(s) to evaluate
   Evaluate the point(s) generated using all computational resources
   Update the Surrogate model
```

Outputs: Best solution and its corresponding function value

Typically used stopping criteria are a maximum number of allowed function evaluations (used in this toolbox), a maximum allowed CPU time, or a maximum number of failed iterative improvement trials.

CHAPTER 3

Options

Optimization problem

The optimization problem is its own object and must have certain attributes and methods in order to work with the framework. We start by giving an example of a mixed-integer optimization problem with constraints. The following attributes and methods must always be specified in the optimization problem class:

- **Attributes**
  - xlow: Lower bounds for the variables.
  - xup: Upper bounds for the variables.
  - dim: Number of dimensions
  - integer: Specifies the integer variables. If no variables have discrete, set to []
  - continuous: Specifies the continuous variables. If no variables are continuous, set to []
- **Required methods**
  - objfunction: Takes one input in the form of an numpy.ndarray with shape (1, dim), which corresponds to one point in dim dimensions. Returns the value (a scalar) of the objective function at this point.
- **Optional methods**
  - eval_ineq_constraints: Only necessary if there are inequality constraints. All constraints must be inequality constraints and the must be written in the form $g_i(x) \leq 0$. The function takes one input in the form of an numpy.ndarray of shape (n, dim), which corresponds to n points in dim dimensions. Returns an numpy.ndarray of shape (n, M) where M is the number of inequality constraints.
  - deriv_ineq_constraints: Only necessary if there are inequality constraints and an adaptive sampling method that requires gradient information of the constraints is used. Returns a numpy ndarray of shape (n, nconstraints, dim)

What follows is an example of an objective function in 5 dimensions with 3 integer and 2 continuous variables. There are also 3 inequality constraints that are not bound constraints which means that we need to implement the eval_ineq_constraints method.

```
import numpy as np

class LinearMI:
    def __init__(self):
        self.xlow = np.zeros(5)
        self.xup = np.array([10, 10, 10, 1, 1])
        self.dim = 5
        self.min = -1
        self.integer = np.arange(0, 3)
        self.continuous = np.arange(3, 5)

    def eval_ineq_constraints(self, x):
        vec = np.zeros((x.shape[0], 3))
        vec[:, 0] = x[:, 0] + x[:, 2] - 1.6
        vec[:, 1] = 1.333 * x[:, 1] + x[:, 3] - 3
        vec[:, 2] = - x[:, 2] - x[:, 3] + x[:, 4]
        return vec

    def objfunction(self, x):
        if len(x) != self.dim:
            raise ValueError('Dimension mismatch')
        return - x[0] + 3 * x[1] + 1.5 * x[2] + 2 * x[3] - 0.5 * x[4]
```

Note: The method check_opt_prob which is available in pySOT is helpful in order t o test that the objective function is compatible with the framework.

Experimental design

The experimental design generates the initial points to be evaluated. A well-chosen experimental design is critical in order to fit a surrogate model that captures the behavior of the underlying objective function. Any implementation must have the following attributes and method:

- **Attributes:**
  - dim: Dimensionality
  - npts: Number of points in the design
- **Required methods**
  - generate_points(): Returns an experimental design of size npts x d where npts is the number of points in the initial design, which was specified when the object was created.

The following experimental designs are supported:

- **LatinHypercube:** A Latin hypercube design

  Example:

  ```
  from pySOT import LatinHypercube
  exp_des = LatinHypercube(dim=3, npts=10)
  ```

  creates a Latin hypercube design with 10 points in 3 dimensions

- **SymmetricLatinHypercube** A symmetric Latin hypercube design

  Example:

```
from pySOT import SymmetricLatinHypercube
exp_des = SymmetricLatinHypercube(dim=3, npts=10)
```

creates a symmetric Latin hypercube design with 10 points in 3 dimensions

- **TwoFactorial** The corners of the unit hypercube

  Example:

```
from pySOT import TwoFactorial
exp_des = TwoFactorial(dim=3)
```

  creates a symmetric Latin hypercube design with 8 points in 3 dimensions

- **BoxBehnken.** Box-Behnken design with one center point. This means that the design consits of the midpoints of the edges of the unit hypercube plus the center of the unit hypercube.

  Example:

```
from pySOT import BoxBehnken
exp_des = BoxBehnken(dim=3)
```

  creates a Box-Behnken design with 13 points in 3 dimensions.

Surrogate model

The surrogate model approximates the underlying objective function given all of the points that have been evaluated. Any implementation of a surrogate model must have the following attributes and methods

- **Attributes:**
  - nump: Number of data points (integer)
  - maxp: Maximum number of data points (integer)
- **Required methods**
  - reset(): Resets the surrogate model
  - get_x(): Returns a numpy array of size nump x d of the data points
  - get_fx(): Returns a numpy array of length nump with the function values
  - add_point(x, f): Adds a point x with value f to the surrogate model
  - eval(x): Evaluates the surrogate model at one point x
  - evals(x): Evaluates the surrogate model at multiple points
- **Optional methods**
  - deriv(x): Returns a numpy array with the gradient at one point x

The following surrogate models are supported:

- **RBFInterpolant:** A radial basis function interpolant.

  Example:

```
from pySOT import RBFInterpolant, CubicKernel, LinearTail
fhat = RBFInterpolant(kernel=CubicKernel, tail=LinearTail, maxp=500)
```

creates a cubic RBF with a linear tail with a capacity for 500 points.

- **GPRegression:** Generate a Gaussian process regression object.

  Note: This implementation depends on the scikit-learn of version 0.18.1 or higher (see *Dependencies*)

  Example:

```
from pySOT import GPRegression
fhat = GPRegression(maxp=500)
```

  creates a GPRegression object with a capacity of 500 points.

- **MARSInterpolant:** Generate a Multivariate Adaptive Regression Splines (MARS) model.

  Note: This implementation depends on the py-earth module (see *Dependencies*)

  Example:

```
from pySOT import MARSInterpolant
fhat = MARSInterpolant(maxp=500)
```

  creates a MARS interpolant with a capacity of 500 points.

- **PolyRegression:** Multivariate polynomial regression.

  Example:

```
from pySOT import PolyRegression
bounds = bounds = np.hstack((np.zeros((3,1)), np.ones((3,1)))) # Our points␣
↪are in [0,1]^3
basisp = basis_TD(3, 2)  # use degree 2 with cross-terms
fhat = PolyRegression(bounds=bounds, basisp=basisp, maxp=500)
```

  creates a polynomial regression surface of degree 2 with no cross-terms interpolant and a capacity of 500 points.

- **EnsembleSurrogate:** We also provide the option of using multiple surrogates for the same problem. Suppose we have M surrogate models, then the ensemble surrogate takes the form

$$s(x) = \sum_{j=1}^{M} w_j s_j(x)$$

  where w_j are non-negative weights that sum to 1. Hence the value of the ensemble surrogate is the weighted prediction of the M surrogate models. We use leave-one-out for each surrogate model to predict the function value at the removed point and then compute several statistics such as correlation with the true function values, RMSE, etc. Based on these statistics we use Dempster-Shafer Theory to compute the pignistic probability for each model, and take this probability as the weight. Surrogate models that does a good job predicting the removed points will generally be given a large weight.

  Example:

```
from pySOT import RBFInterpolant, CubicKernel, LinearTail, \
                  GPRegression, MARSInterpolant, EnsembleSurrogate

models = [
```

```
    RBFInterpolant(kernel=CubicKernel, tail=LinearTail, maxp=500), \
    GPRegression(maxp=500), MARSInterpolant(maxp=500)
]

response_surface = EnsembleSurrogate(model_list=models, maxp=500)
```

creates an ensemble surrogate with three surrogate models, namely a Cubic RBF Interpolant, a MARS interpolant, and a Gaussian process regression object.

Adaptive sampling

We provide several different methods for selecting the next point to evaluate. All methods in this version are based in generating candidate points by perturbing the best solution found so far or in some cases just choose a random point. We also provide the option of using many different strategies in the same experiment and how to cycle between the different strategies. Each implementation of this object is required to have the following attributes and methods

- **Attributes:**
  - proposed_points: List of points proposed to the optimization algorithm
- **Required methods**
  - init(start_sample, fhat, budget): This initializes the sampling strategy by providing the points that were evaluated in the experimental design phase, the response surface, and also provides the evaluation budget.
  - remove_point(x): Removes point x from list of proposed_points if the evaluation crashed or was never carried out by the strategy. Returns True if the point was removed and False if the removal failed.
  - make_points(npts, xbest, sigma, subset=None, proj_fun=None): This is the method that proposes npts new evaluations to the strategy. It needs to know the number of points to propose, the best data point evaluated so far, the preferred sample radius of the strategy (w.r.t the unit box), the coordinates that the strategy wants to perturb, and a way to project points onto the feasible region.

We now list the different options and describe shortly how they work.

- **CandidateSRBF:** Generate perturbations around the best solution found so far
- **CandidateSRBF_INT:** Uses CandidateSRBF but only perturbs the integer variables
- **CandidateSRBF_CONT:** Uses CandidateSRBF but only perturbs the continuous variables
- **CandidateDYCORS:** Uses a DYCORS strategy which perturbs each coordinate with some iteration dependent probability. This probability is a monotonically decreasing function with the number of iteration.
- **CandidateDYCORS_CONT:** Uses CandidateDYCORS but only perturbs the continuous variables
- **CandidateDYCORS_INT:** Uses CandidateDYCORS but only perturbs the integer variables
- **CandidateDDS:** Uses the DDS strategy where only a few candidate points are generated and the one with the best surrogate prediction is picked for evaluation
- **CandidateDDS_CONT:** Uses CandidateDDS but only perturbs the continuous variables
- **CandidateDDS_INT:** Uses CandidateDDS but only perturbs the integer variables
- **CandidateUniform:** Chooses a new point uniformly from the box-constrained domain
- **CandidateUniform_CONT:** Given the best solution found so far the continuous variables are chosen uniformly from the box-constrained domain

- **CandidateUniform_INT:** Given the best solution found so far the integer variables are chosen uniformly from the box-constrained domain

The CandidateDYCORS algorithm is the bread-and-butter algorithm for any problems with more than 5 dimensions whilst CandidateSRBF is recommended for problems with only a few dimensions. It is sometimes efficient in mixed-integer problems to perturb the integer and continuous variables separately and we therefore provide such method for each of these algorithms. Finally, uniformly choosing a new point has the advantage of creating diversity to avoid getting stuck in a local minima. Each method needs an objective function object as described in the previous section (the input name is data) and how many perturbations should be generated around the best solution found so far (the input name is numcand). Around 100 points per dimension, but no more than 5000, is recommended. Next is an example on how to generate a multi-start strategy that uses CandidateDYCORS, CandidateDYCORS_CONT, CandidateDYCORS_INT, and CandidateUniform and that cycles evenly between the methods i.e., the first point is generated using CandidateDYCORS, the second using CandidateDYCORS_CONT and so on.

```
from pySOT import LinearMI, MultiSampling, CandidateDYCORS, \
                  CandidateDYCORS_CONT, CandidateDYCORS_INT, \
                  CandidateUniform

data = LinearMI()  # Optimization problem
sampling_methods = [CandidateDYCORS(data=data, numcand=100*data.dim), \
                    CandidateDYCORS_CONT(data=data, numcand=100*data.dim), \
                    CandidateDYCORS_INT(data=data, numcand=100*data.dim), \
                    CandidateUniform(data=data, numcand=100*data.dim)]
cycle = [0, 1, 2, 3]
sampling_methods = MultiSampling(sampling_methods, cycle)
```

CHAPTER 4

POAP

pySOT uses POAP, which an event-driven framework for building and combining asynchronous optimization strategies. There are two main components in POAP, namely controllers and strategies. The controller is capable of asking workers to run function evaluations and the strategy decides where to evaluate next. POAP works with external black-box objective functions and handles potential crashes in the objective function evaluation. There is also a logfile from which all function evaluations can be accessed after the run finished. In its simplest form, an optimization code with POAP that evaluates a function predetermined set of points using NUM_WORKERS threads may look the following way:

```
from poap.strategy import FixedSampleStrategy
from poap.strategy import CheckWorkStrategy
from poap.controller import ThreadController
from poap.controller import BasicWorkerThread

# samples = list of sample points ...

controller = ThreadController()
sampler = FixedSampleStrategy(samples)
controller.strategy = CheckWorkerStrategy(controller, sampler)

for i in range(NUM_WORKERS):
   t = BasicWorkerThread(controller, objective)
   controller.launch_worker(t)

result = controller.run()
print 'Best result: {0} at {1}'.format(result.value, result.params)
```

Controller

The controller is responsible for accepting or rejecting proposals by the strategy object, controlling and monitoring the workers, and informing the strategy object of relevant events. Examples of relevant events are the processing of a proposal, or status updates on a function evaluation. Interactions between controller and the strategies are organized around proposals and evaluation records. At the beginning of the optimization and on any later change to the system

state, the controller requests a proposal from the strategy. The proposal consists of an action (evaluate a function, kill a function, or terminate the optimization), a list of parameters, and a list of callback functions to be executed once the proposal is processed. The controller then either accepts the proposal (and sends a command to the worker), or rejects the proposal.

When the controller accepts a proposal to start a function evaluation, it creates an evaluation record to share information about the status of the evaluation with the strategy. The evaluation record includes the evaluation point, the status of the evaluation, the value (if completed), and a list of callback functions to be executed on any update. Once a proposal has been accepted or rejected, the controller processes any pending system events (e.g. completed or canceled function evaluations), notifies the strategy about updates, and requests the next proposed action.

POAP comes with a serial controller which is the controller of choice when objective function evaluations are carried out in serial. There is also a threaded controller that dispatches work to a queue of workers where each worker is able to handle evaluation and kill requests. The requests are asynchronous in the sense that the workers are not required to complete the evaluation or termination requests. The worker is forced to respond to evaluation requests, but may ignore kill requests. When receiving an evaluation request, the worker should either attempt the evaluation or mark the record as killed. The worker sends status updates back to the controller by updating the relevant record. There is also a third controller that uses simulated time, which is very useful for testing asynchronous optimization strategies.

Strategy

The strategy is the heart of the optimization algorithm, since it is responsible for choosing new evaluations, killing evaluations, and terminating the optimization run when a stopping criteria is reached. POAP provides some basic default strategies based on non-adaptive sampling and serial optimization routines and also some strategies that adapt or combine other strategies.

Different strategies can be composed by combining their control actions, which can be used to let a strategy cycle through a list of optimization strategies and select the most promising of their proposals. Strategies can also subscribe to be informed of all new function evaluations so they incorporate any new function information, even though the evaluation was proposed by another strategy. This makes it possible to start several independent strategies while still allowing each strategy to look at the function information that comes from function evaluations proposed by other strategies. As an example we can have a local optimizer strategy running a gradient based method where the starting point can be selected based on the best point found by any other strategy. The flexibility of the POAP framework makes combined strategies like these very straightforward.

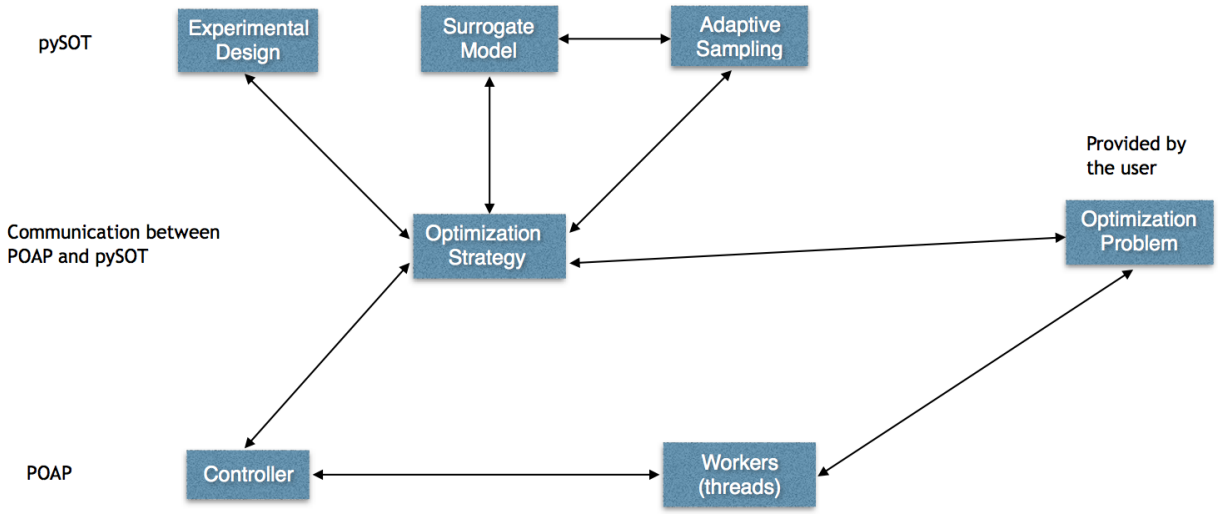
Workers

The multi-threaded controller employs a set of workers that are capable of managing concurrent function evaluations. Each worker does not provide parallelism on its own, but the worker itself is allowed to exploit parallelism by separate external processes.

There are workers that are capable of calling Python objective function when asked to do an evaluation, which only results in parallelism if the objective function implementation itself allows parallelism. There are workers that use subprocesses in order to carry out external objective function evaluations that are not necessarily in Python. The user is responsible for specifying how to evaluate the objective function and how to parse partial information if available.

POAP is also capable of having workers connect to a specified TCP/IP port in order to communicate with the controller. This functionality is useful in a cluster setting, for example, where the workers should run on compute nodes distinct from the node where the controller is running. It is also very useful in a setting where the workers run on a supercomputer that has a restriction on the number of hours per job submission. Having the controller run on a separate machine will allow the controller to keep running and the workers to reconnect and continue carrying out evaluations.

Communication between POAP and pySOT



CHAPTER 5

Graphical user interface

pySOT comes with a graphical user interface (GUI) built in PyQt5. In order to use the GUI you need to have PyQt5 installed together with all other dependencies of pySOT. Initializing the GUI is as easy as typing from the terminal:

```
from pySOT.gui import GUI
GUI()
```

or more compactly:

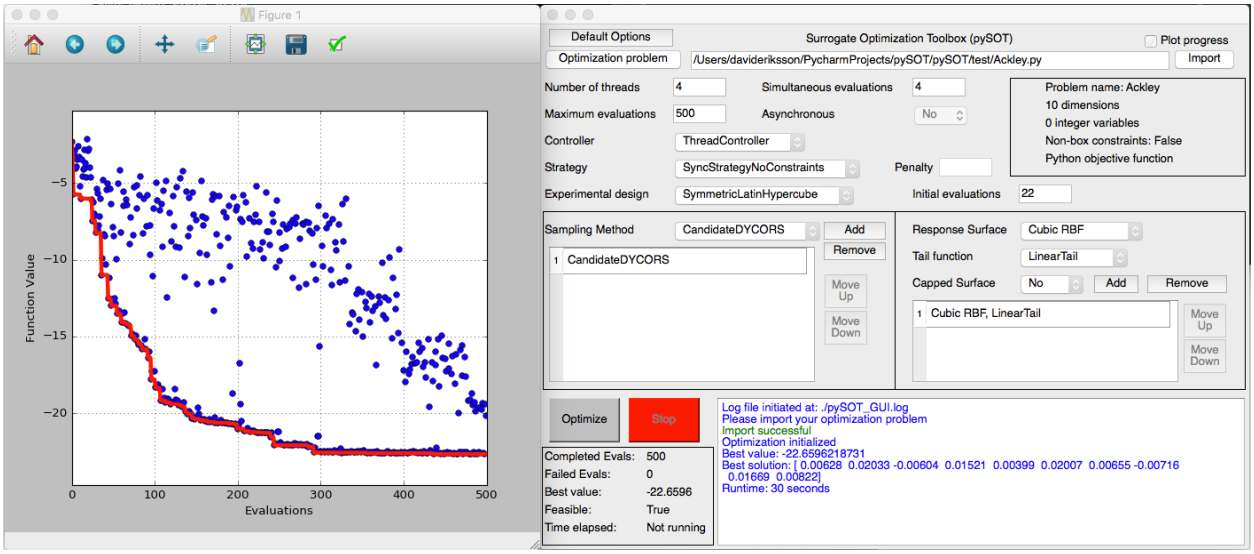
```
python -c 'from pySOT.gui import GUI; GUI()'
```

The optimization problem has to be implemented in a separate file and this file must satisfy the requirements mentioned above for an optimization problem. In addition, the separate Python implementation is only allowed to contain one class and this class has to have the same name as the file name (excluding .py). As an example, this is an implementation of the Ackley function in a separate file with file name Ackley.py:

```
import numpy as np

class Ackley:
    def __init__(self, dim=10):
        self.xlow = -15 * np.ones(dim)
        self.xup = 20 * np.ones(dim)
        self.dim = dim
        self.info = str(dim)+"-dimensional Ackley function \n" +\
                            "Global optimum: f(0,0,...,0) = 0"
        self.integer = []
        self.continuous = np.arange(0, dim)

    def objfunction(self, x):
        if len(x) != self.dim:
            raise ValueError('Dimension mismatch')
        n = float(len(x))
        return -20.0 * np.exp(-0.2*np.sqrt(sum(x**2)/n)) - \
                       np.exp(sum(np.cos(2.0*np.pi*x))/n)
```

Note that both the file name and the class names are the same.

The picture below shows what the GUI looks like.



CHAPTER 6

Tutorials

What follows are 7 pySOT tutorials that are based on a short course given at CMWR 2016, Toronto. These notebooks are available at: https://people.cam.cornell.edu/~dme65/talks.html

Tutorial 1: Hello World!

First example to show how to use pySOT in serial and synchronous parallel for bound constrained optimization problems

Step 1: Import modules and create pySOT objects (1)-(4)

```
# Import the necessary modules
from pySOT import *
from poap.controller import SerialController, ThreadController, BasicWorkerThread
import numpy as np

# Decide how many evaluations we are allowed to use
maxeval = 500

# (1) Optimization problem
# Use the 10-dimensional Ackley function
data = Ackley(dim=10)
print(data.info)

# (2) Experimental design
# Use a symmetric Latin hypercube with 2d + 1 samples
exp_des = SymmetricLatinHypercube(dim=data.dim, npts=2*data.dim+1)

# (3) Surrogate model
# Use a cubic RBF interpolant with a linear tail
surrogate = RBFInterpolant(kernel=CubicKernel, tail=LinearTail, maxp=maxeval)

# (4) Adaptive sampling
```

```
# Use DYCORS with 100d candidate points
adapt_samp = CandidateDYCORS(data=data, numcand=100*data.dim)
```

Step 2: Launch a serial controller and use standard Surrogate Optimization strategy

```
# Use the serial controller (uses only one thread)
controller = SerialController(data.objfunction)

# (5) Use the sychronous strategy without non-bound constraints
strategy = SyncStrategyNoConstraints(
        worker_id=0, data=data, maxeval=maxeval, nsamples=1,
        exp_design=exp_des, response_surface=surrogate,
        sampling_method=adapt_samp)
controller.strategy = strategy

# Run the optimization strategy
result = controller.run()

# Print the final result
print('Best value found: {0}'.format(result.value))
print('Best solution found: {0}'.format(
    np.array_str(result.params[0], max_line_width=np.inf,
                 precision=5, suppress_small=True)))
```

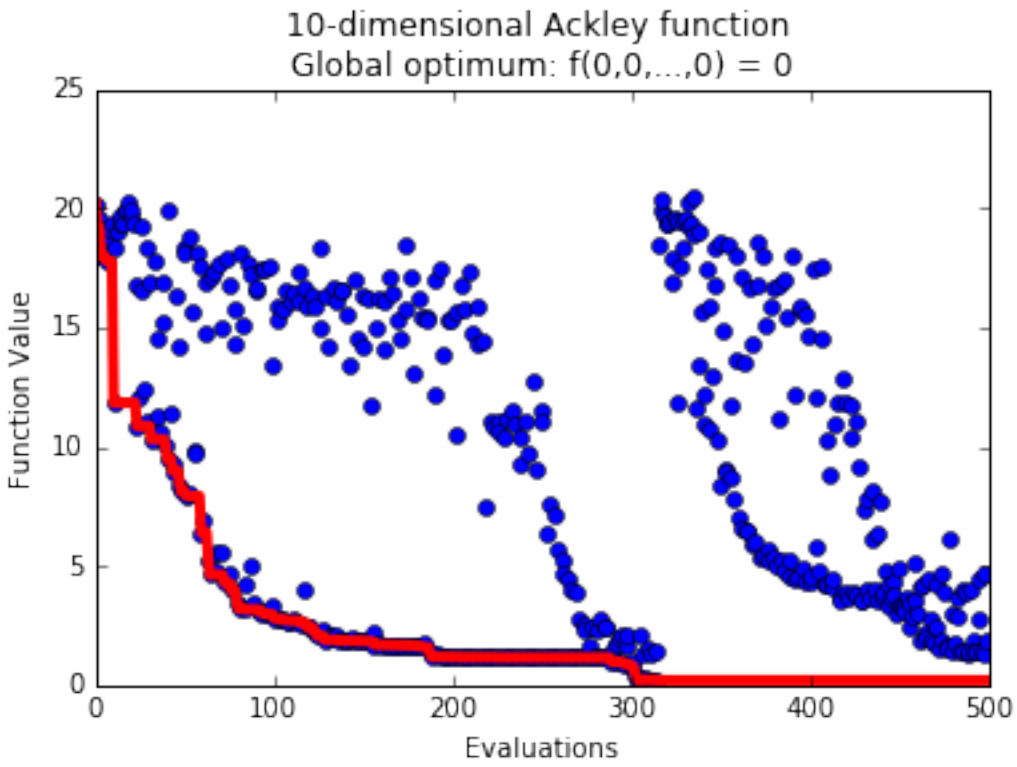
Possible output:

```
Best value found: 0.211036185111
Best solution found: [ 0.02193  0.00486  0.03323  0.03656 -0.00228 -0.00414  0.05239 -
↪0.08511 -0.0002   0.00104]
```

Step 3: Make a progress plot

```
import matplotlib.pyplot as plt

# Extract function values from the controller
fvals = np.array([o.value for o in controller.fevals])

f, ax = plt.subplots()
ax.plot(np.arange(0,maxeval), fvals, 'bo')  # Points
ax.plot(np.arange(0,maxeval), np.minimum.accumulate(fvals), 'r-', linewidth=4.0)  #␣
↪Best value found
plt.xlabel('Evaluations')
plt.ylabel('Function Value')
plt.title(data.info)
plt.show()
```

Possible output:



Step 4: Launch a threaded controller with 4 workers and use standard Surrogate Optimization strategy allowing to do 4 simultaneous in parallel. Notice how similar the code in Step 3 is to the code in Step 2.

```
# Use the threaded controller
controller = ThreadController()

# (5) Use the sychronous strategy without non-bound constraints
# Use 4 threads and allow for 4 simultaneous evaluations
nthreads = 4
strategy = SyncStrategyNoConstraints(
        worker_id=0, data=data, maxeval=maxeval, nsamples=nthreads,
        exp_design=exp_des, response_surface=surrogate,
        sampling_method=adapt_samp)
controller.strategy = strategy

# Launch the threads and give them access to the objective function
for _ in range(nthreads):
    worker = BasicWorkerThread(controller, data.objfunction)
    controller.launch_worker(worker)

# Run the optimization strategy
result = controller.run()

# Print the final result
print('Best value found: {0}'.format(result.value))
print('Best solution found: {0}'.format(
    np.array_str(result.params[0], max_line_width=np.inf,
                 precision=5, suppress_small=True)))
```

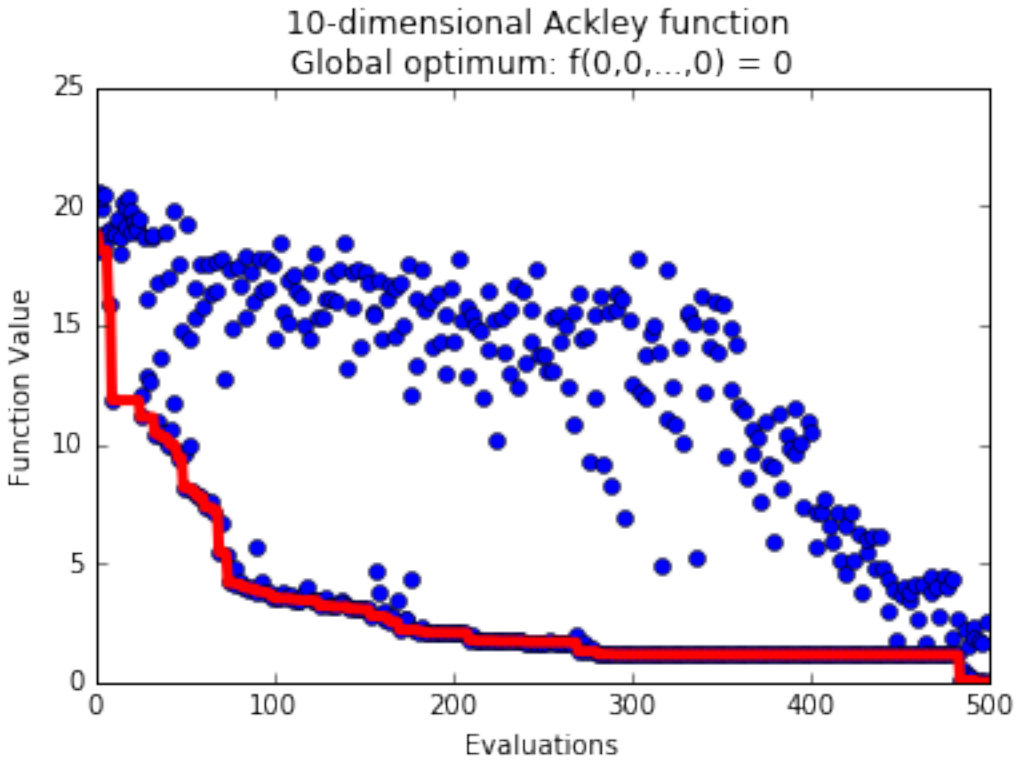
Possible output:

```
Best value found: 0.0143986694788
Best solution found: [-0.00869 -0.00018 -0.00311 -0.00211  0.00098  0.00161  0.00219 -
↪0.00241  0.00285  0.00255]
```

Step 5 Make a progress plot

```
import matplotlib.pyplot as plt

# Extract function values from the controller
fvals = np.array([o.value for o in controller.fevals])

f, ax = plt.subplots()
ax.plot(np.arange(0,maxeval), fvals, 'bo')  # Points
ax.plot(np.arange(0,maxeval), np.minimum.accumulate(fvals), 'r-', linewidth=4.0)  #␣
↪Best value found
plt.xlabel('Evaluations')
plt.ylabel('Function Value')
plt.title(data.info)
plt.show()
```

Possible output:



Tutorial 2: Python objective function

This example shows how to define our own optimization problem in pySOT

Step 1: Define our own optimization problem

```
class SomeFun:
    def __init__(self, dim=10):
        self.xlow = -10 * np.ones(dim) # lower bounds
        self.xup = 10 * np.ones(dim) # upper bounds
        self.dim = dim # dimensionality
        self.info = "Our own " + str(dim)+"-dimensional function" # info
        self.integer = np.array([0]) # integer variables
        self.continuous = np.arange(1, dim) # continuous variables

    def objfunction(self, x):
        return np.sum(x) * np.cos(np.sum(x))
```

Step 2: Let's make sure that our optimization problem follows the pySOT standard

```
import numpy as np
from pySOT import check_opt_prob
data = SomeFun(dim=10)
check_opt_prob(data)
```

Everything is fine as long as pySOT doesn't complain!

Step 3: Import modules and create pySOT objects (1)-(4)

```
# Import the necessary modules
from pySOT import *
from poap.controller import SerialController, BasicWorkerThread

# Decide how many evaluations we are allowed to use
maxeval = 500

# (1) Optimization problem
# Use our 10-dimensional function
print(data.info)

# (2) Experimental design
# Use a symmetric Latin hypercube with 2d + 1 samples
exp_des = SymmetricLatinHypercube(dim=data.dim, npts=2*data.dim+1)

# (3) Surrogate model
# Use a cubic RBF interpolant with a linear tail
surrogate = RBFInterpolant(kernel=CubicKernel, tail=LinearTail, maxp=maxeval)

# (4) Adaptive sampling
# Use DYCORS with 100d candidate points
adapt_samp = CandidateDYCORS(data=data, numcand=100*data.dim)
```

Output:

```
Our own 10-dimensional function
```

Step 4: Run the optimization in serial

```
# Use the serial controller (uses only one thread)
controller = SerialController(data.objfunction)

# (5) Use the sychronous strategy without non-bound constraints
strategy = SyncStrategyNoConstraints(
        worker_id=0, data=data, maxeval=maxeval, nsamples=1,
```

```
        exp_design=exp_des, response_surface=surrogate,
        sampling_method=adapt_samp)
controller.strategy = strategy

# Run the optimization strategy
result = controller.run()

# Print the final result
print('Best value found: {0}'.format(result.value))
print('Best solution found: {0}'.format(
    np.array_str(result.params[0], max_line_width=np.inf,
                 precision=5, suppress_small=True)))
```

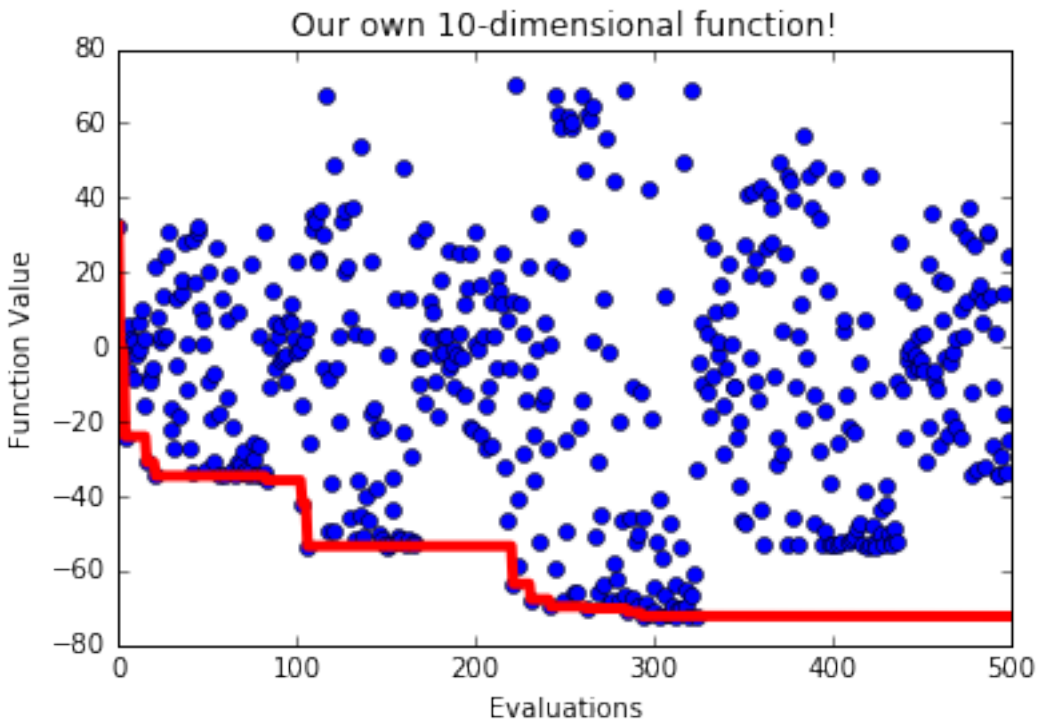
Possible output:

```
Best value found: -72.2440613978
Best solution found: [ 9.       5.58049  9.34501  5.35848  9.26448  9.05695  5.45796
↪1.80559  8.16331  9.21498]
```

Step 5: Plot the progress:

```
import matplotlib.pyplot as plt

# Extract function values from the controller
fvals = np.array([o.value for o in controller.fevals])

f, ax = plt.subplots()
ax.plot(np.arange(0,maxeval), fvals, 'bo')  # Points
ax.plot(np.arange(0,maxeval), np.minimum.accumulate(fvals), 'r-', linewidth=4.0)  #
↪Best value found
plt.xlabel('Evaluations')
plt.ylabel('Function Value')
plt.title(data.info)
plt.show()
```

Possible output:



Tutorial 3: MATLAB objective function

This example shows how to use pySOT with an objective function that is written in MATLAB. You need the matlab_wrapper module or the MATLAB engine which is available for MATLAB R2014b or later. This example uses the matlab_wrapper module to work for older versions of MATLAB as well.

Step 1: The following shows an implementation of the Ackley function in a MATLAB script matlab_ackley.m that takes a variable x from the workspace and saves the value of the objective function as val:

```
dim = length(x);
val = -20*exp(-0.2*sqrt(sum(x.^2,2)/dim)) - ...
  exp(sum(cos(2*pi*x),2)/dim) + 20 + exp(1);
```

Step 2: This will create a MATLAB session. You may need to specify the root folder of your MATLAB installation. Type matlabroot in a MATLAB session to see what the root folder is.

```
import matlab_wrapper
matlab = matlab_wrapper.MatlabSession(matlab_root='/Applications/MATLAB_R2014a.app',␣
↪options='-nojvm')
```

Step 3: Define Python optimization problem that uses our MATLAB objective function to do function evaluations:

```
# This is the path to the external MATLAB function, assuming it is in your current␣
↪path
import os
mfile_location = os.getcwd()
matlab.workspace.addpath(mfile_location)

class AckleyExt:
    def __init__(self, dim=10):
```

```
        self.xlow = -15 * np.ones(dim)
        self.xup = 20 * np.ones(dim)
        self.dim = dim
        self.info = str(dim) + "-dimensional Ackley function \n" + \
            "Global optimum: f(0,0,...,0) = 0"
        self.min = 0
        self.integer = []
        self.continuous = np.arange(0, dim)

    def objfunction(self, x):
        matlab.put('x', x)
        matlab.eval('matlab_ackley')
        val = matlab.get('val')
        return val
```

Step 4: Optimize over our optimization problem

```
from pySOT import *
from poap.controller import SerialController, BasicWorkerThread
import numpy as np

maxeval = 500

data = AckleyExt(dim=10)
print(data.info)

# Use the serial controller for simplicity
# In order to run in parallel we need to maintain an array of MATLAB session
controller = SerialController(data.objfunction)
controller.strategy = \
    SyncStrategyNoConstraints(
        worker_id=0, data=data,
        maxeval=maxeval, nsamples=1,
        exp_design=LatinHypercube(dim=data.dim, npts=2*(data.dim+1)),
        response_surface=RBFInterpolant(kernel=CubicKernel, tail=LinearTail,␣
↪maxp=maxeval),
        sampling_method=CandidateDYCORS(data=data, numcand=100*data.dim))

# Run the optimization strategy
result = controller.run()

# Print the final result
print('Best value found: {0}'.format(result.value))
print('Best solution found: {0}'.format(
    np.array_str(result.params[0], max_line_width=np.inf,
                 precision=5, suppress_small=True)))
```

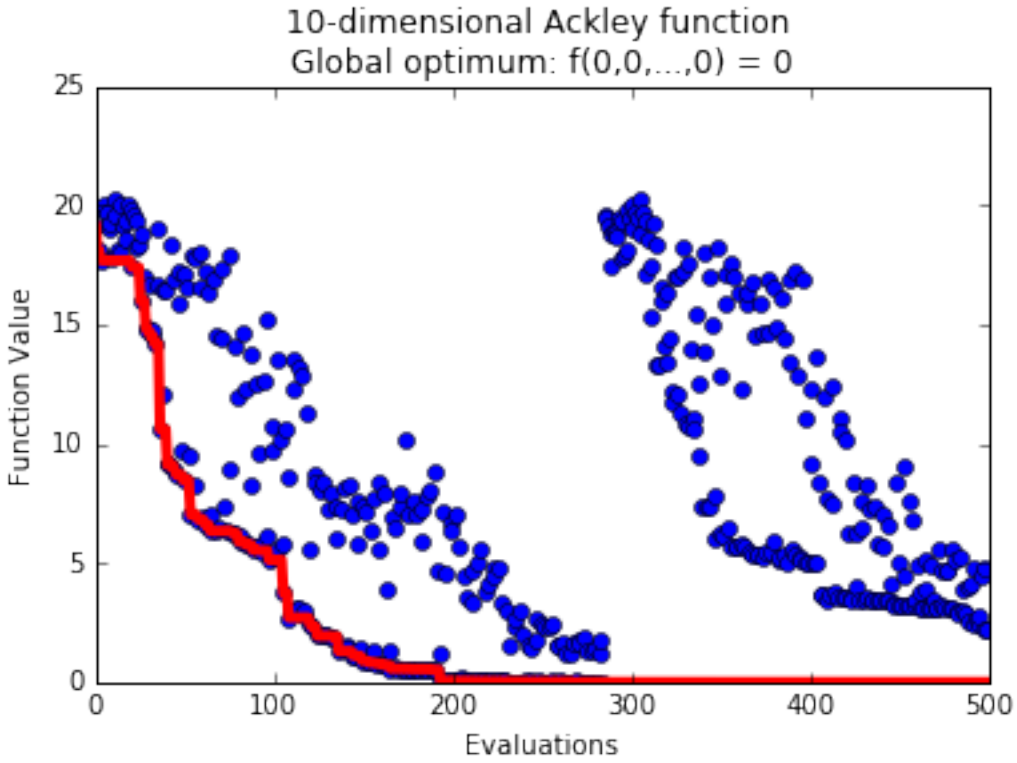
Possible output:

```
10-dimensional Ackley function
Global optimum: f(0,0,...,0) = 0
Best value found: 0.00665167450159
Best solution found: [-0.00164  0.00162 -0.00122  0.0019  -0.00109  0.00197 -0.00102 -
↪0.00124 -0.00194  0.00216]
```

Step 5: Plot the progress:

```
import matplotlib.pyplot as plt

# Extract function values from the controller
fvals = np.array([o.value for o in controller.fevals])

f, ax = plt.subplots()
ax.plot(np.arange(0,maxeval), fvals, 'bo')  # Points
ax.plot(np.arange(0,maxeval), np.minimum.accumulate(fvals), 'r-', linewidth=4.0)  #␣
↪Best value found
plt.xlabel('Evaluations')
plt.ylabel('Function Value')
plt.title(data.info)
plt.show()
```

Possible output:



Step 6: End the MATLAB session:

```
matlab.__del__()
```

Note: The example test_matlab_engine.py in pySOT.test shows how to use a MATLAB engine with more than 1 worker. The main idea is the give each worker its own MATLAB session that the worker can do for function evaluations.

Tutorial 4: Python objective function with inequality constraints

This example considers the Keane bump function which has two inequality constraints and takes the following form:

$$f(x_1, \ldots, x_d) = -\left| \frac{\sum_{j=1}^{d} \cos^4(x_j) - 2\prod_{j=1}^{d} \cos^2(x_j)}{\sqrt{\sum_{j=1}^{d} j x_j^2}} \right|$$

subject to:

$$0 \leq x_i \leq 5$$

$$0.75 - \prod_{j=1}^{d} x_j < 0$$

$$\sum_{j=1}^{d} x_j - 7.5d < 0$$

The global optimum approaches -0.835 when d goes to infinity. We will use pySOT and the penalty method approach to optimize over the Keane bump function and we will use 4 workers in synchronous parallel. The code that achieves this is

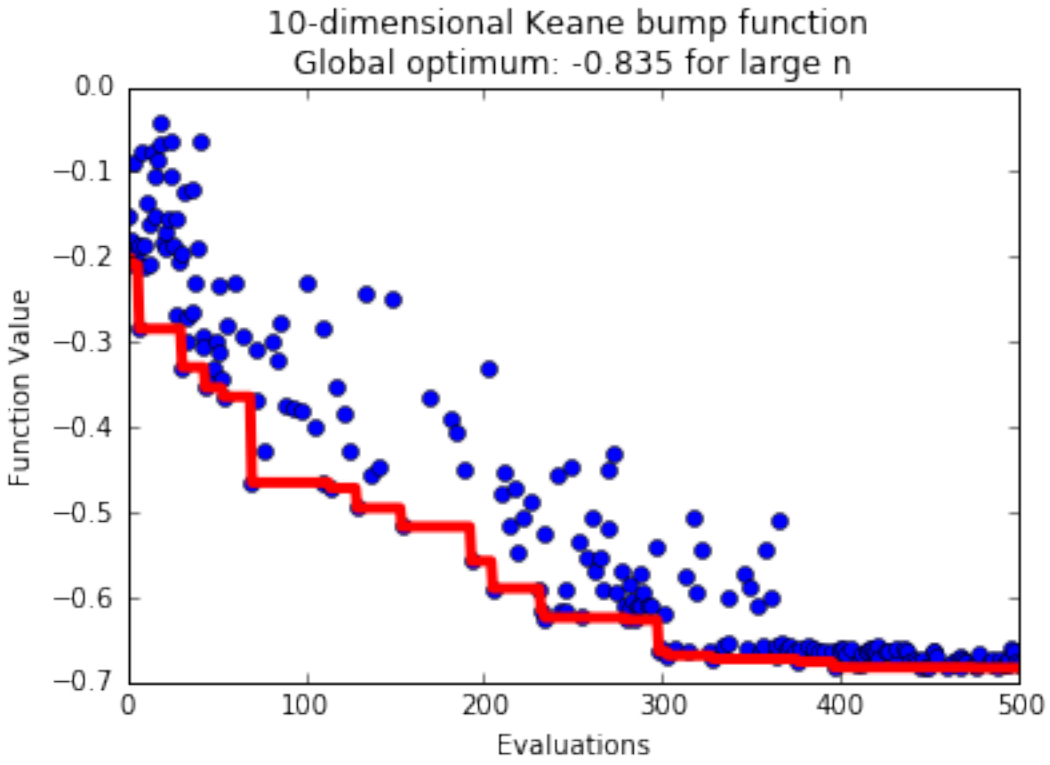
```
from pySOT import *
from poap.controller import Threaded, BasicWorkerThread
import numpy as np

maxeval = 500

data = Keane(dim=10)
print(data.info)

controller = ThreadController()

# Use 4 threads and allow for 4 simultaneous evaluations
nthreads = 4
strategy = SyncStrategyPenalty(
        worker_id=0, data=data,
        maxeval=maxeval, nsamples=1,
        exp_design=LatinHypercube(dim=data.dim, npts=2*(data.dim+1)),
        response_surface=RBFInterpolant(kernel=CubicKernel, tail=LinearTail,␣
↪maxp=maxeval),
        sampling_method=CandidateDYCORS(data=data, numcand=100*data.dim))
controller.strategy = strategy

# Launch the threads and give them access to the objective function
for _ in range(nthreads):
    worker = BasicWorkerThread(controller, data.objfunction)
    controller.launch_worker(worker)

# Returns f(x) is feasible, infinity otherwise
def feasible_merit(record):
    return record.value if record.feasible else np.inf

# Run the optimization strategy and ask the controller for the best FEASIBLE solution
result = controller.run(merit=feasible_merit)
best, xbest = result.value, result.params[0]
```

```
# Print the final result
print('Best value found: {0}'.format(result.value))
print('Best solution found: {0}'.format(
    np.array_str(result.params[0], max_line_width=np.inf,
                 precision=5, suppress_small=True)))

# Check constraints
print('\nConstraint 1: 0.75 - prod(x) = {0}'.format(0.75 - np.prod(xbest)))
print('Constraint 2: sum(x) - 7.5*dim = {0}'.format(np.sum(xbest) - 7.5*data.dim))
```

Possible output:

```
Best value found: -0.683081148607
Best solution found: [ 3.11277  3.07498  2.91834  2.96004  2.84659  1.29008  0.17825
↪0.31923  0.19628  0.24831]

Constraint 1: 0.75 - prod(x) = -0.0921329885647
Constraint 2: sum(x) - 7.5*dim = -57.8551318917
```

A possible progress plot is:



Tutorial 5: Equality constraints

The only way pySOT supports inequality constraints is via a projection method. That is, the user needs to supply a method that projects any infeasible point onto the feasible region. This is trivial in some cases such as when we have a normalization constraint of the form $g(x) = 1 - \|x\| = 0$, in which case we can just rescale each infeasible point. The purpose of this example is to show how to use pySOT for such a constraint and we will modify the Ackley function by adding a constraint that the solution needs to have unit 2-norm. Our new objective function takes the form

```
class AckleyUnit:
    def __init__(self, dim=10):
        self.xlow = -1 * np.ones(dim)
        self.xup = 1 * np.ones(dim)
        self.dim = dim
        self.info = str(dim)+"-dimensional Ackley function on the unit sphere \n" +\
                              "Global optimum: f(1,0,...,0) = ... = f(0,0,...,1) = " +\
                              str(np.round(20*(1-np.exp(-0.2/np.sqrt(dim))), 3))
        self.min = 20*(1 - np.exp(-0.2/np.sqrt(dim)))
        self.integer = []
        self.continuous = np.arange(0, dim)
        check_opt_prob(self)

    def objfunction(self, x):
        n = float(len(x))
        return -20.0 * np.exp(-0.2*np.sqrt(np.sum(x**2)/n)) - \
            np.exp(np.sum(np.cos(2.0*np.pi*x))/n) + 20 + np.exp(1)

    def eval_eq_constraints(self, x):
        return np.linalg.norm(x) - 1
```

We next define a projection method as follows:

```
import numpy as np

def projection(x):
    return x / np.linalg.norm(x)
```

Optimizing over this function is done via

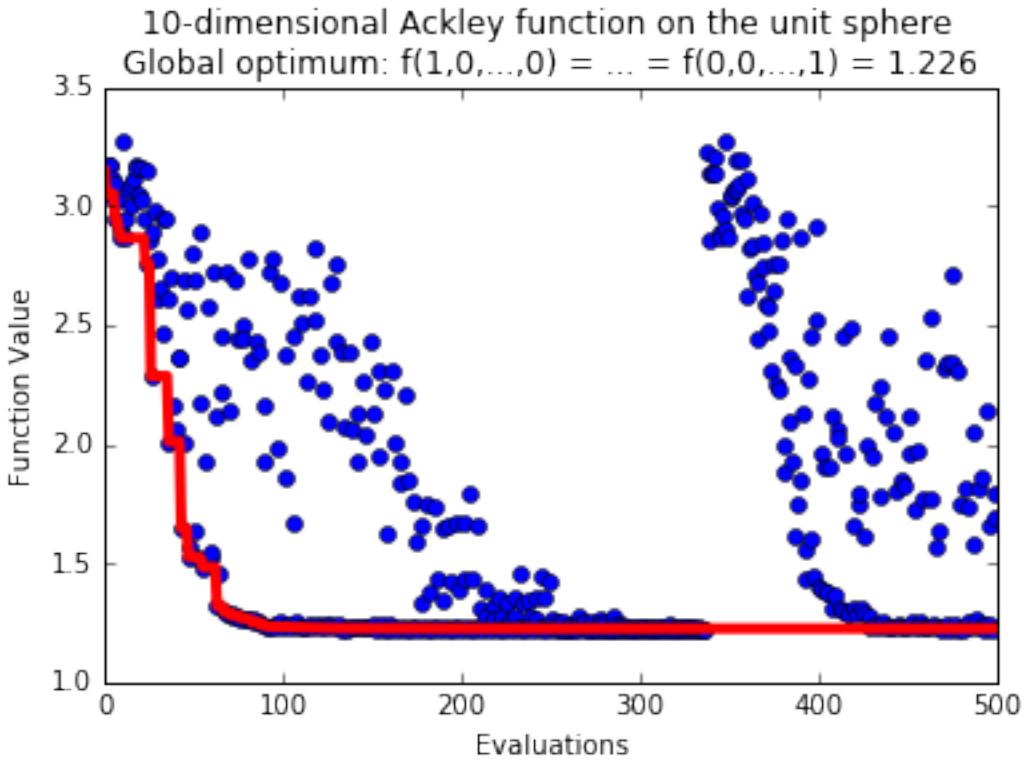
```
from pySOT import *
from poap.controller import Threaded, BasicWorkerThread
import numpy as np

maxeval = 500

data = AckleyUnit(dim=10)
print(data.info)

controller = ThreadController()

# Use 4 threads and allow for 4 simultaneous evaluations
nthreads = 4
strategy = SyncStrategyProjection(
        worker_id=0, data=data,
        maxeval=maxeval, nsamples=1,
        exp_design=LatinHypercube(dim=data.dim, npts=2*(data.dim+1)),
        response_surface=RBFInterpolant(kernel=CubicKernel, tail=LinearTail,␣
↪maxp=maxeval),
        sampling_method=CandidateDYCORS(data=data, numcand=100*data.dim),
        proj_fun=projection)
controller.strategy = strategy

# Launch the threads and give them access to the objective function
for _ in range(nthreads):
    worker = BasicWorkerThread(controller, data.objfunction)
    controller.launch_worker(worker)
```

```
# Run the optimization strategy and ask the controller for the best FEASIBLE solution
result = controller.run()

# Print the final result
print('Best value found: {0}'.format(result.value))
print('Best solution found: {0}'.format(
    np.array_str(result.params[0], max_line_width=np.inf,
                 precision=5, suppress_small=True)))

# Check constraint
print('\n ||x|| = {0}'.format(np.linalg.norm(result.params[0])))
```

Possible output:

```
Best value found: 1.22580826108
Best solution found: [-0.00017  0.00106  0.00172 -0.00126  0.0013  -0.00035  0.00133
↪0.99999 -0.00114  0.00138]
||x|| = 1.0
```

A possible progress plot if the following:



Tutorial 6: C++ objective function

Stay patient!

CHAPTER 7

Guidelines

Stay patient!

CHAPTER 8

Logging

pySOT logs all important events that occur during the optimization process. The user can specify what level of logging he wants to do. The five levels are:

- critical
- error
- warning
- info
- debug

Function evaluations are recorded on the info level, so this is the recommended level for pySOT. There is currently nothing that is being logged on the debug level, but better logging for debugging will likely be added in the future. Crashed evaluations are recorded on the warning level.

More information about logging in Python 2.7 is available at: https://docs.python.org/2/library/logging.html.

CHAPTER 9

Source code

pySOT.adaptive_sampling module

Module adaptive_sampling

Author David Eriksson <dme65@cornell.edu>, David Bindel <bindel@cornell.edu>

class pySOT.adaptive_sampling.**CandidateDDS**(*data*, *numcand=None*, *weights=None*)
An implementation of the DDS candidate points method

Only a few candidate points are generated and the candidate point with the lowest value predicted by the surrogate model is selected. The DDS method only perturbs a subset of the dimensions when perturbing the best solution. The probability for a dimension to be perturbed decreases after each evaluation and is capped in order to guarantee global convergence.

Parameters

- **data** (*Object*) – Optimization problem object
- **numcand** (*int*) – Number of candidate points to be used. Default is min([5000, 100*data.dim])
- **weights** (*list of numpy.array*) – Weights used for the merit function, to balance exploration vs exploitation

Raises **ValueError** – If number of candidate points is incorrect or if the weights aren't a list in [0, 1]

Variables

- **data** – Optimization problem object
- **fhat** – Response surface object
- **xrange** – Variable ranges, xup - xlow
- **dtol** – Smallest allowed distance between evaluated points 1e-3 * sqrt(dim)
- **weights** – Weights used for the merit function

- **proposed_points** – List of points proposed to the optimization algorithm
- **dmerit** – Minimum distance between the points and the proposed points
- **xcand** – Candidate points
- **fhvals** – Predicted values by the surrogate model
- **next_weight** – Index of the next weight to be used
- **numcand** – Number of candidate points
- **budget** – Remaining evaluation budget
- **probfun** – Function that computes the perturbation probability of a given iteration

Note: This object needs to be initialized with the init method. This is done when the initial phase has finished.

Todo

Get rid of the proposed_points object and replace it by something that is controlled by the strategy.

init(*start_sample*, *fhat*, *budget*)
Initialize the sampling method after the initial phase

This initializes the list of sampling methods after the initial phase has finished and the experimental design has been evaluated. The user provides the points in the experimental design, the surrogate model, and the remaining evaluation budget.

Parameters

- **start_sample** (*numpy.array*) – Points in the experimental design
- **fhat** (*Object*) – Surrogate model
- **budget** (*int*) – Evaluation budget

make_points(*npts*, *xbest*, *sigma*, *subset=None*, *proj_fun=None*, *merit=<function candidate_merit_weighted_distance>*)
Proposes npts new points to evaluate

Parameters

- **npts** (*int*) – Number of points to select
- **xbest** (*numpy.array*) – Best solution found so far
- **sigma** (*float*) – Current sampling radius w.r.t the unit box
- **subset** (*numpy.array*) – Coordinates to perturb, the others are fixed
- **proj_fun** (*Object*) – Routine for projecting infeasible points onto the feasible region
- **merit** (*Object*) – Merit function for selecting candidate points

Returns Points selected for evaluation, of size npts x dim

Return type numpy.array

Todo

Change the merit function from being hard-coded

remove_point(*x*)
Remove x from proposed_points

This removes x from the list of proposed points in the case where the optimization strategy decides to not evaluate x.

Parameters **x** (*numpy.array*) – Point to be removed

Returns True if points was removed, False otherwise

Type bool

class pySOT.adaptive_sampling.**CandidateDDS_CONT**(*data*, *numcand=None*, *weights=None*)
CandidateDDS where only the the continuous variables are perturbed

Parameters

- **data** (*Object*) – Optimization problem object
- **numcand** (*int*) – Number of candidate points to be used. Default is min([5000, 100*data.dim])
- **weights** (*list of numpy.array*) – Weights used for the merit function, to balance exploration vs exploitation

Raises **ValueError** – If number of candidate points is incorrect or if the weights aren't a list in [0, 1]

Variables

- **data** – Optimization problem object
- **fhat** – Response surface object
- **xrange** – Variable ranges, xup - xlow
- **dtol** – Smallest allowed distance between evaluated points 1e-3 * sqrt(dim)
- **weights** – Weights used for the merit function
- **proposed_points** – List of points proposed to the optimization algorithm
- **dmerit** – Minimum distance between the points and the proposed points
- **xcand** – Candidate points
- **fhvals** – Predicted values by the surrogate model
- **next_weight** – Index of the next weight to be used
- **numcand** – Number of candidate points
- **budget** – Remaining evaluation budget
- **probfun** – Function that computes the perturbation probability of a given iteration

Note: This object needs to be initialized with the init method. This is done when the initial phase has finished.

Todo

Get rid of the proposed_points object and replace it by something that is controlled by the strategy.

init(*start_sample*, *fhat*, *budget*)
Initialize the sampling method after the initial phase

This initializes the list of sampling methods after the initial phase has finished and the experimental design has been evaluated. The user provides the points in the experimental design, the surrogate model, and the remaining evaluation budget.

Parameters

- **start_sample** (*numpy.array*) – Points in the experimental design
- **fhat** (*Object*) – Surrogate model
- **budget** (*int*) – Evaluation budget

make_points(*npts*, *xbest*, *sigma*, *subset=None*, *proj_fun=None*, *merit=<function candidate_merit_weighted_distance>*)
Proposes npts new points to evaluate

Parameters

- **npts** (*int*) – Number of points to select
- **xbest** (*numpy.array*) – Best solution found so far
- **sigma** (*float*) – Current sampling radius w.r.t the unit box
- **subset** (*numpy.array*) – Coordinates to perturb, the others are fixed
- **proj_fun** (*Object*) – Routine for projecting infeasible points onto the feasible region
- **merit** (*Object*) – Merit function for selecting candidate points

Returns Points selected for evaluation, of size npts x dim

Return type numpy.array

Todo

Change the merit function from being hard-coded

---

remove_point(*x*)
Remove x from proposed_points

This removes x from the list of proposed points in the case where the optimization strategy decides to not evaluate x.

Parameters **x** (*numpy.array*) – Point to be removed

Returns True if points was removed, False otherwise

Type bool

class pySOT.adaptive_sampling.**CandidateDDS_INT**(*data*, *numcand=None*, *weights=None*)
CandidateDDS where only the the integer variables are perturbed

Parameters

- **data** (*Object*) – Optimization problem object
- **numcand** (*int*) – Number of candidate points to be used. Default is min([5000, 100*data.dim])
- **weights** (*list of numpy.array*) – Weights used for the merit function, to balance exploration vs exploitation

Raises **ValueError** – If number of candidate points is incorrect or if the weights aren't a list in [0, 1]

Variables

- **data** – Optimization problem object
- **fhat** – Response surface object
- **xrange** – Variable ranges, xup - xlow
- **dtol** – Smallest allowed distance between evaluated points 1e-3 * sqrt(dim)
- **weights** – Weights used for the merit function
- **proposed_points** – List of points proposed to the optimization algorithm
- **dmerit** – Minimum distance between the points and the proposed points
- **xcand** – Candidate points
- **fhvals** – Predicted values by the surrogate model
- **next_weight** – Index of the next weight to be used
- **numcand** – Number of candidate points
- **budget** – Remaining evaluation budget
- **probfun** – Function that computes the perturbation probability of a given iteration

---

Note: This object needs to be initialized with the init method. This is done when the initial phase has finished.

Todo

Get rid of the proposed_points object and replace it by something that is controlled by the strategy.

---

init(*start_sample*, *fhat*, *budget*)

Initialize the sampling method after the initial phase

This initializes the list of sampling methods after the initial phase has finished and the experimental design has been evaluated. The user provides the points in the experimental design, the surrogate model, and the remaining evaluation budget.

Parameters

- **start_sample** (*numpy.array*) – Points in the experimental design
- **fhat** (*Object*) – Surrogate model
- **budget** (*int*) – Evaluation budget

make_points(*npts*, *xbest*, *sigma*, *subset=None*, *proj_fun=None*, *merit=<function candidate_merit_weighted_distance>*)

Proposes npts new points to evaluate

Parameters

- **npts** (*int*) – Number of points to select
- **xbest** (*numpy.array*) – Best solution found so far
- **sigma** (*float*) – Current sampling radius w.r.t the unit box
- **subset** (*numpy.array*) – Coordinates to perturb, the others are fixed

- **proj_fun** (*Object*) – Routine for projecting infeasible points onto the feasible region
- **merit** (*Object*) – Merit function for selecting candidate points

Returns Points selected for evaluation, of size npts x dim

Return type numpy.array

Todo

Change the merit function from being hard-coded

---

remove_point(*x*)

Remove x from proposed_points

This removes x from the list of proposed points in the case where the optimization strategy decides to not evaluate x.

Parameters **x** (*numpy.array*) – Point to be removed

Returns True if points was removed, False otherwise

Type bool

class pySOT.adaptive_sampling.**CandidateDYCORS**(*data*, *numcand=None*, *weights=None*)

An implementation of the DYCORS method

The DYCORS method only perturbs a subset of the dimensions when perturbing the best solution. The probability for a dimension to be perturbed decreases after each evaluation and is capped in order to guarantee global convergence.

Parameters

- **data** (*Object*) – Optimization problem object
- **numcand** (*int*) – Number of candidate points to be used. Default is min([5000, 100*data.dim])
- **weights** (*list of numpy.array*) – Weights used for the merit function, to balance exploration vs exploitation

Raises **ValueError** – If number of candidate points is incorrect or if the weights aren't a list in [0, 1]

Variables

- **data** – Optimization problem object
- **fhat** – Response surface object
- **xrange** – Variable ranges, xup - xlow
- **dtol** – Smallest allowed distance between evaluated points 1e-3 * sqrt(dim)
- **weights** – Weights used for the merit function
- **proposed_points** – List of points proposed to the optimization algorithm
- **dmerit** – Minimum distance between the points and the proposed points
- **xcand** – Candidate points
- **fhvals** – Predicted values by the surrogate model
- **next_weight** – Index of the next weight to be used

- **numcand** – Number of candidate points
- **budget** – Remaining evaluation budget
- **minprob** – Smallest allowed perturbation probability
- **n0** – Evaluations spent when the initial phase ended
- **probfun** – Function that computes the perturbation probability of a given iteration

Note: This object needs to be initialized with the init method. This is done when the initial phase has finished.

Todo

Get rid of the proposed_points object and replace it by something that is controlled by the strategy.

init(*start_sample*, *fhat*, *budget*)
Initialize the sampling method after the initial phase

This initializes the list of sampling methods after the initial phase has finished and the experimental design has been evaluated. The user provides the points in the experimental design, the surrogate model, and the remaining evaluation budget.

Parameters

- **start_sample** (*numpy.array*) – Points in the experimental design
- **fhat** (*Object*) – Surrogate model
- **budget** (*int*) – Evaluation budget

make_points(*npts*, *xbest*, *sigma*, *subset=None*, *proj_fun=None*, *merit=<function candidate_merit_weighted_distance>*)
Proposes npts new points to evaluate

Parameters

- **npts** (*int*) – Number of points to select
- **xbest** (*numpy.array*) – Best solution found so far
- **sigma** (*float*) – Current sampling radius w.r.t the unit box
- **subset** (*numpy.array*) – Coordinates to perturb, the others are fixed
- **proj_fun** (*Object*) – Routine for projecting infeasible points onto the feasible region
- **merit** (*Object*) – Merit function for selecting candidate points

Returns Points selected for evaluation, of size npts x dim

Return type numpy.array

Todo

Change the merit function from being hard-coded

remove_point(*x*)
Remove x from proposed_points

This removes x from the list of proposed points in the case where the optimization strategy decides to not evaluate x.

Parameters **x** (*numpy.array*) – Point to be removed

Returns True if points was removed, False otherwise

Type bool

class pySOT.adaptive_sampling.**CandidateDYCORS_CONT**(*data*, *numcand=None*, *weights=None*)

CandidateDYCORS where only the the continuous variables are perturbed

Parameters

- **data** (*Object*) – Optimization problem object
- **numcand** (*int*) – Number of candidate points to be used. Default is min([5000, 100*data.dim])
- **weights** (*list of numpy.array*) – Weights used for the merit function, to balance exploration vs exploitation

Raises **ValueError** – If number of candidate points is incorrect or if the weights aren't a list in [0, 1]

Variables

- **data** – Optimization problem object
- **fhat** – Response surface object
- **xrange** – Variable ranges, xup - xlow
- **dtol** – Smallest allowed distance between evaluated points 1e-3 * sqrt(dim)
- **weights** – Weights used for the merit function
- **proposed_points** – List of points proposed to the optimization algorithm
- **dmerit** – Minimum distance between the points and the proposed points
- **xcand** – Candidate points
- **fhvals** – Predicted values by the surrogate model
- **next_weight** – Index of the next weight to be used
- **numcand** – Number of candidate points
- **budget** – Remaining evaluation budget
- **probfun** – Function that computes the perturbation probability of a given iteration

---

Note: This object needs to be initialized with the init method. This is done when the initial phase has finished.

Todo

Get rid of the proposed_points object and replace it by something that is controlled by the strategy.

---

init(*start_sample*, *fhat*, *budget*)

Initialize the sampling method after the initial phase

This initializes the list of sampling methods after the initial phase has finished and the experimental design has been evaluated. The user provides the points in the experimental design, the surrogate model, and the remaining evaluation budget.

Parameters

- **start_sample** (*numpy.array*) – Points in the experimental design
- **fhat** (*Object*) – Surrogate model
- **budget** (*int*) – Evaluation budget

make_points(*npts*, *xbest*, *sigma*, *subset=None*, *proj_fun=None*, *merit=<function candidate_merit_weighted_distance>*)

Proposes npts new points to evaluate

Parameters

- **npts** (*int*) – Number of points to select
- **xbest** (*numpy.array*) – Best solution found so far
- **sigma** (*float*) – Current sampling radius w.r.t the unit box
- **subset** (*numpy.array*) – Coordinates to perturb, the others are fixed
- **proj_fun** (*Object*) – Routine for projecting infeasible points onto the feasible region
- **merit** (*Object*) – Merit function for selecting candidate points

Returns Points selected for evaluation, of size npts x dim

Return type numpy.array

Todo

Change the merit function from being hard-coded

---

remove_point(*x*)

Remove x from proposed_points

This removes x from the list of proposed points in the case where the optimization strategy decides to not evaluate x.

Parameters **x** (*numpy.array*) – Point to be removed

Returns True if points was removed, False otherwise

Type bool

class pySOT.adaptive_sampling.**CandidateDYCORS_INT**(*data*, *numcand=None*, *weights=None*)

CandidateDYCORS where only the the integer variables are perturbed

Parameters

- **data** (*Object*) – Optimization problem object
- **numcand** (*int*) – Number of candidate points to be used. Default is min([5000, 100*data.dim])
- **weights** (*list of numpy.array*) – Weights used for the merit function, to balance exploration vs exploitation

Raises **ValueError** – If number of candidate points is incorrect or if the weights aren't a list in [0, 1]

Variables

- **data** – Optimization problem object
- **fhat** – Response surface object

- **xrange** – Variable ranges, xup - xlow
- **dtol** – Smallest allowed distance between evaluated points 1e-3 * sqrt(dim)
- **weights** – Weights used for the merit function
- **proposed_points** – List of points proposed to the optimization algorithm
- **dmerit** – Minimum distance between the points and the proposed points
- **xcand** – Candidate points
- **fhvals** – Predicted values by the surrogate model
- **next_weight** – Index of the next weight to be used
- **numcand** – Number of candidate points
- **budget** – Remaining evaluation budget
- **probfun** – Function that computes the perturbation probability of a given iteration

---

Note: This object needs to be initialized with the init method. This is done when the initial phase has finished.

Todo

Get rid of the proposed_points object and replace it by something that is controlled by the strategy.

---

init(*start_sample*, *fhat*, *budget*)

Initialize the sampling method after the initial phase

This initializes the list of sampling methods after the initial phase has finished and the experimental design has been evaluated. The user provides the points in the experimental design, the surrogate model, and the remaining evaluation budget.

Parameters

- **start_sample** (*numpy.array*) – Points in the experimental design
- **fhat** (*Object*) – Surrogate model
- **budget** (*int*) – Evaluation budget

make_points(*npts*, *xbest*, *sigma*, *subset=None*, *proj_fun=None*, *merit=<function candidate_merit_weighted_distance>*)

Proposes npts new points to evaluate

Parameters

- **npts** (*int*) – Number of points to select
- **xbest** (*numpy.array*) – Best solution found so far
- **sigma** (*float*) – Current sampling radius w.r.t the unit box
- **subset** (*numpy.array*) – Coordinates to perturb, the others are fixed
- **proj_fun** (*Object*) – Routine for projecting infeasible points onto the feasible region
- **merit** (*Object*) – Merit function for selecting candidate points

Returns Points selected for evaluation, of size npts x dim

Return type numpy.array

Todo

Change the merit function from being hard-coded

remove_point(*x*)
Remove x from proposed_points

This removes x from the list of proposed points in the case where the optimization strategy decides to not evaluate x.

Parameters **x** (*numpy.array*) – Point to be removed

Returns True if points was removed, False otherwise

Type bool

class pySOT.adaptive_sampling.**CandidateSRBF**(*data*, *numcand=None*, *weights=None*)
An implementation of Stochastic RBF

This is an implementation of the candidate points method that is proposed in the first SRBF paper. Candidate points are generated by making normally distributed perturbations with standard deviation sigma around the best solution. The candidate point that minimizes a specified merit function is selected as the next point to evaluate.

Parameters

- **data** (*Object*) – Optimization problem object
- **numcand** (*int*) – Number of candidate points to be used. Default is min([5000, 100*data.dim])
- **weights** (*list of numpy.array*) – Weights used for the merit function, to balance exploration vs exploitation

Raises **ValueError** – If number of candidate points is incorrect or if the weights aren't a list in [0, 1]

Variables

- **data** – Optimization problem object
- **fhat** – Response surface object
- **xrange** – Variable ranges, xup - xlow
- **dtol** – Smallest allowed distance between evaluated points 1e-3 * sqrt(dim)
- **weights** – Weights used for the merit function
- **proposed_points** – List of points proposed to the optimization algorithm
- **dmerit** – Minimum distance between the points and the proposed points
- **xcand** – Candidate points
- **fhvals** – Predicted values by the surrogate model
- **next_weight** – Index of the next weight to be used
- **numcand** – Number of candidate points
- **budget** – Remaining evaluation budget

Note: This object needs to be initialized with the init method. This is done when the initial phase has finished.

Todo

Get rid of the proposed_points object and replace it by something that is controlled by the strategy.

init(*start_sample*, *fhat*, *budget*)
Initialize the sampling method after the initial phase

This initializes the list of sampling methods after the initial phase has finished and the experimental design has been evaluated. The user provides the points in the experimental design, the surrogate model, and the remaining evaluation budget.

Parameters

- **start_sample** (*numpy.array*) – Points in the experimental design
- **fhat** (*Object*) – Surrogate model
- **budget** (*int*) – Evaluation budget

make_points(*npts*, *xbest*, *sigma*, *subset=None*, *proj_fun=None*, *merit=<function candidate_merit_weighted_distance>*)
Proposes npts new points to evaluate

Parameters

- **npts** (*int*) – Number of points to select
- **xbest** (*numpy.array*) – Best solution found so far
- **sigma** (*float*) – Current sampling radius w.r.t the unit box
- **subset** (*numpy.array*) – Coordinates to perturb, the others are fixed
- **proj_fun** (*Object*) – Routine for projecting infeasible points onto the feasible region
- **merit** (*Object*) – Merit function for selecting candidate points

Returns Points selected for evaluation, of size npts x dim

Return type numpy.array

Todo

Change the merit function from being hard-coded

remove_point(*x*)
Remove x from proposed_points

This removes x from the list of proposed points in the case where the optimization strategy decides to not evaluate x.

Parameters **x** (*numpy.array*) – Point to be removed

Returns True if points was removed, False otherwise

Type bool

class pySOT.adaptive_sampling.**CandidateSRBF_CONT**(*data*, *numcand=None*, *weights=None*)
CandidateSRBF where only the the continuous variables are perturbed

Parameters

- **data** (*Object*) – Optimization problem object

- **numcand** (*int*) – Number of candidate points to be used. Default is min([5000, 100*data.dim])
- **weights** (*list of numpy.array*) – Weights used for the merit function, to balance exploration vs exploitation

Raises **ValueError** – If number of candidate points is incorrect or if the weights aren't a list in [0, 1]

Variables

- **data** – Optimization problem object
- **fhat** – Response surface object
- **xrange** – Variable ranges, xup - xlow
- **dtol** – Smallest allowed distance between evaluated points 1e-3 * sqrt(dim)
- **weights** – Weights used for the merit function
- **proposed_points** – List of points proposed to the optimization algorithm
- **dmerit** – Minimum distance between the points and the proposed points
- **xcand** – Candidate points
- **fhvals** – Predicted values by the surrogate model
- **next_weight** – Index of the next weight to be used
- **numcand** – Number of candidate points
- **budget** – Remaining evaluation budget
- **probfun** – Function that computes the perturbation probability of a given iteration

---

Note: This object needs to be initialized with the init method. This is done when the initial phase has finished.

Todo

Get rid of the proposed_points object and replace it by something that is controlled by the strategy.

---

init(*start_sample*, *fhat*, *budget*)

Initialize the sampling method after the initial phase

This initializes the list of sampling methods after the initial phase has finished and the experimental design has been evaluated. The user provides the points in the experimental design, the surrogate model, and the remaining evaluation budget.

Parameters

- **start_sample** (*numpy.array*) – Points in the experimental design
- **fhat** (*Object*) – Surrogate model
- **budget** (*int*) – Evaluation budget

make_points(*npts*, *xbest*, *sigma*, *subset=None*, *proj_fun=None*, *merit=<function candidate_merit_weighted_distance>*)

Proposes npts new points to evaluate

Parameters

- **npts** (*int*) – Number of points to select
- **xbest** (*numpy.array*) – Best solution found so far
- **sigma** (*float*) – Current sampling radius w.r.t the unit box
- **subset** (*numpy.array*) – Coordinates to perturb, the others are fixed
- **proj_fun** (*Object*) – Routine for projecting infeasible points onto the feasible region
- **merit** (*Object*) – Merit function for selecting candidate points

Returns Points selected for evaluation, of size npts x dim

Return type numpy.array

Todo

Change the merit function from being hard-coded

---

remove_point(*x*)

Remove x from proposed_points

This removes x from the list of proposed points in the case where the optimization strategy decides to not evaluate x.

Parameters **x** (*numpy.array*) – Point to be removed

Returns True if points was removed, False otherwise

Type bool

class pySOT.adaptive_sampling.**CandidateSRBF_INT**(*data*, *numcand=None*, *weights=None*)

CandidateSRBF where only the the integer variables are perturbed

Parameters

- **data** (*Object*) – Optimization problem object
- **numcand** (*int*) – Number of candidate points to be used. Default is min([5000, 100*data.dim])
- **weights** (*list of numpy.array*) – Weights used for the merit function, to balance exploration vs exploitation

Raises **ValueError** – If number of candidate points is incorrect or if the weights aren't a list in [0, 1]

Variables

- **data** – Optimization problem object
- **fhat** – Response surface object
- **xrange** – Variable ranges, xup - xlow
- **dtol** – Smallest allowed distance between evaluated points 1e-3 * sqrt(dim)
- **weights** – Weights used for the merit function
- **proposed_points** – List of points proposed to the optimization algorithm
- **dmerit** – Minimum distance between the points and the proposed points
- **xcand** – Candidate points
- **fhvals** – Predicted values by the surrogate model

- **next_weight** – Index of the next weight to be used
- **numcand** – Number of candidate points
- **budget** – Remaining evaluation budget
- **probfun** – Function that computes the perturbation probability of a given iteration

Note: This object needs to be initialized with the init method. This is done when the initial phase has finished.

Todo

Get rid of the proposed_points object and replace it by something that is controlled by the strategy.

init(*start_sample*, *fhat*, *budget*)

Initialize the sampling method after the initial phase

This initializes the list of sampling methods after the initial phase has finished and the experimental design has been evaluated. The user provides the points in the experimental design, the surrogate model, and the remaining evaluation budget.

Parameters

- **start_sample** (*numpy.array*) – Points in the experimental design
- **fhat** (*Object*) – Surrogate model
- **budget** (*int*) – Evaluation budget

make_points(*npts*, *xbest*, *sigma*, *subset=None*, *proj_fun=None*, *merit=<function candidate_merit_weighted_distance>*)

Proposes npts new points to evaluate

Parameters

- **npts** (*int*) – Number of points to select
- **xbest** (*numpy.array*) – Best solution found so far
- **sigma** (*float*) – Current sampling radius w.r.t the unit box
- **subset** (*numpy.array*) – Coordinates to perturb, the others are fixed
- **proj_fun** (*Object*) – Routine for projecting infeasible points onto the feasible region
- **merit** (*Object*) – Merit function for selecting candidate points

Returns Points selected for evaluation, of size npts x dim

Return type numpy.array

Todo

Change the merit function from being hard-coded

remove_point(*x*)

Remove x from proposed_points

This removes x from the list of proposed points in the case where the optimization strategy decides to not evaluate x.

Parameters **x** (*numpy.array*) – Point to be removed

Returns True if points was removed, False otherwise

Type bool

class pySOT.adaptive_sampling.**CandidateUniform**(*data*, *numcand=None*, *weights=None*)

Create Candidate points by sampling uniformly in the domain

Parameters

- **data** (*Object*) – Optimization problem object
- **numcand** (*int*) – Number of candidate points to be used. Default is min([5000, 100*data.dim])
- **weights** (*list of numpy.array*) – Weights used for the merit function, to balance exploration vs exploitation

Raises **ValueError** – If number of candidate points is incorrect or if the weights aren't a list in [0, 1]

Variables

- **data** – Optimization problem object
- **fhat** – Response surface object
- **xrange** – Variable ranges, xup - xlow
- **dtol** – Smallest allowed distance between evaluated points 1e-3 * sqrt(dim)
- **weights** – Weights used for the merit function
- **proposed_points** – List of points proposed to the optimization algorithm
- **dmerit** – Minimum distance between the points and the proposed points
- **xcand** – Candidate points
- **fhvals** – Predicted values by the surrogate model
- **next_weight** – Index of the next weight to be used
- **numcand** – Number of candidate points
- **budget** – Remaining evaluation budget

Note: This object needs to be initialized with the init method. This is done when the initial phase has finished.

Todo

Get rid of the proposed_points object and replace it by something that is controlled by the strategy.

---

init(*start_sample*, *fhat*, *budget*)

Initialize the sampling method after the initial phase

This initializes the list of sampling methods after the initial phase has finished and the experimental design has been evaluated. The user provides the points in the experimental design, the surrogate model, and the remaining evaluation budget.

Parameters

- **start_sample** (*numpy.array*) – Points in the experimental design
- **fhat** (*Object*) – Surrogate model

- **budget** (*int*) – Evaluation budget

make_points(*npts*, *xbest*, *sigma*, *subset=None*, *proj_fun=None*, *merit=<function candidate_merit_weighted_distance>*)

Proposes npts new points to evaluate

Parameters

- **npts** (*int*) – Number of points to select
- **xbest** (*numpy.array*) – Best solution found so far
- **sigma** (*float*) – Current sampling radius w.r.t the unit box
- **subset** (*numpy.array*) – Coordinates to perturb, the others are fixed
- **proj_fun** (*Object*) – Routine for projecting infeasible points onto the feasible region
- **merit** (*Object*) – Merit function for selecting candidate points

Returns Points selected for evaluation, of size npts x dim

Return type numpy.array

Todo

Change the merit function from being hard-coded

---

remove_point(*x*)

Remove x from proposed_points

This removes x from the list of proposed points in the case where the optimization strategy decides to not evaluate x.

Parameters **x** (*numpy.array*) – Point to be removed

Returns True if points was removed, False otherwise

Type bool

class pySOT.adaptive_sampling.**CandidateUniform_CONT**(*data*, *numcand=None*, *weights=None*)

CandidateUniform where only the the continuous variables are perturbed

Parameters

- **data** (*Object*) – Optimization problem object
- **numcand** (*int*) – Number of candidate points to be used. Default is min([5000, 100*data.dim])
- **weights** (*list of numpy.array*) – Weights used for the merit function, to balance exploration vs exploitation

Raises **ValueError** – If number of candidate points is incorrect or if the weights aren't a list in [0, 1]

Variables

- **data** – Optimization problem object
- **fhat** – Response surface object
- **xrange** – Variable ranges, xup - xlow
- **dtol** – Smallest allowed distance between evaluated points 1e-3 * sqrt(dim)

- **weights** – Weights used for the merit function
- **proposed_points** – List of points proposed to the optimization algorithm
- **dmerit** – Minimum distance between the points and the proposed points
- **xcand** – Candidate points
- **fhvals** – Predicted values by the surrogate model
- **next_weight** – Index of the next weight to be used
- **numcand** – Number of candidate points
- **budget** – Remaining evaluation budget
- **probfun** – Function that computes the perturbation probability of a given iteration

Note: This object needs to be initialized with the init method. This is done when the initial phase has finished.

Todo

Get rid of the proposed_points object and replace it by something that is controlled by the strategy.

init(*start_sample*, *fhat*, *budget*)

Initialize the sampling method after the initial phase

This initializes the list of sampling methods after the initial phase has finished and the experimental design has been evaluated. The user provides the points in the experimental design, the surrogate model, and the remaining evaluation budget.

Parameters

- **start_sample** (*numpy.array*) – Points in the experimental design
- **fhat** (*Object*) – Surrogate model
- **budget** (*int*) – Evaluation budget

make_points(*npts*, *xbest*, *sigma*, *subset=None*, *proj_fun=None*, *merit=<function candidate_merit_weighted_distance>*)

Proposes npts new points to evaluate

Parameters

- **npts** (*int*) – Number of points to select
- **xbest** (*numpy.array*) – Best solution found so far
- **sigma** (*float*) – Current sampling radius w.r.t the unit box
- **subset** (*numpy.array*) – Coordinates to perturb, the others are fixed
- **proj_fun** (*Object*) – Routine for projecting infeasible points onto the feasible region
- **merit** (*Object*) – Merit function for selecting candidate points

Returns Points selected for evaluation, of size npts x dim

Return type numpy.array

Todo

Change the merit function from being hard-coded

remove_point(*x*)
Remove x from proposed_points

This removes x from the list of proposed points in the case where the optimization strategy decides to not evaluate x.

Parameters **x** (*numpy.array*) – Point to be removed

Returns True if points was removed, False otherwise

Type bool

class pySOT.adaptive_sampling.**CandidateUniform_INT**(*data*, *numcand=None*, *weights=None*)
CandidateUniform where only the the integer variables are perturbed

Parameters

- **data** (*Object*) – Optimization problem object
- **numcand** (*int*) – Number of candidate points to be used. Default is min([5000, 100*data.dim])
- **weights** (*list of numpy.array*) – Weights used for the merit function, to balance exploration vs exploitation

Raises **ValueError** – If number of candidate points is incorrect or if the weights aren't a list in [0, 1]

Variables

- **data** – Optimization problem object
- **fhat** – Response surface object
- **xrange** – Variable ranges, xup - xlow
- **dtol** – Smallest allowed distance between evaluated points 1e-3 * sqrt(dim)
- **weights** – Weights used for the merit function
- **proposed_points** – List of points proposed to the optimization algorithm
- **dmerit** – Minimum distance between the points and the proposed points
- **xcand** – Candidate points
- **fhvals** – Predicted values by the surrogate model
- **next_weight** – Index of the next weight to be used
- **numcand** – Number of candidate points
- **budget** – Remaining evaluation budget
- **probfun** – Function that computes the perturbation probability of a given iteration

Note: This object needs to be initialized with the init method. This is done when the initial phase has finished.

Todo

Get rid of the proposed_points object and replace it by something that is controlled by the strategy.

---

init(*start_sample*, *fhat*, *budget*)
Initialize the sampling method after the initial phase

This initializes the list of sampling methods after the initial phase has finished and the experimental design has been evaluated. The user provides the points in the experimental design, the surrogate model, and the remaining evaluation budget.

Parameters

- **start_sample** (*numpy.array*) – Points in the experimental design
- **fhat** (*Object*) – Surrogate model
- **budget** (*int*) – Evaluation budget

make_points(*npts*, *xbest*, *sigma*, *subset=None*, *proj_fun=None*, *merit=<function candidate_merit_weighted_distance>*)
Proposes npts new points to evaluate

Parameters

- **npts** (*int*) – Number of points to select
- **xbest** (*numpy.array*) – Best solution found so far
- **sigma** (*float*) – Current sampling radius w.r.t the unit box
- **subset** (*numpy.array*) – Coordinates to perturb, the others are fixed
- **proj_fun** (*Object*) – Routine for projecting infeasible points onto the feasible region
- **merit** (*Object*) – Merit function for selecting candidate points

Returns Points selected for evaluation, of size npts x dim

Return type numpy.array

Todo

Change the merit function from being hard-coded

---

remove_point(*x*)
Remove x from proposed_points

This removes x from the list of proposed points in the case where the optimization strategy decides to not evaluate x.

Parameters **x** (*numpy.array*) – Point to be removed

Returns True if points was removed, False otherwise

Type bool

class pySOT.adaptive_sampling.**GeneticAlgorithm**(*data*)
Genetic algorithm for minimizing the surrogate model

Parameters **data** (*Object*) – Optimization problem object

Variables

- **data** – Optimization problem object
- **fhat** – Response surface object

- **xrange** – Variable ranges, xup - xlow
- **dtol** – Smallest allowed distance between evaluated points 1e-3 * sqrt(dim)
- **proposed_points** – List of points proposed to the optimization algorithm
- **budget** – Remaining evaluation budget

Note: This object needs to be initialized with the init method. This is done when the initial phase has finished.

---

init (*start_sample*, *fhat*, *budget*)
Initialize the sampling method after the initial phase

This initializes the list of sampling methods after the initial phase has finished and the experimental design has been evaluated. The user provides the points in the experimental design, the surrogate model, and the remaining evaluation budget.

Parameters

- **start_sample** (*numpy.array*) – Points in the experimental design
- **fhat** (*Object*) – Surrogate model
- **budget** (*int*) – Evaluation budget

make_points (*npts*, *xbest*, *sigma*, *subset=None*, *proj_fun=None*, *merit=None*)
Proposes npts new points to evaluate

Parameters

- **npts** (*int*) – Number of points to select
- **xbest** (*numpy.array*) – Best solution found so far (Ignored)
- **sigma** (*float*) – Current sampling radius w.r.t the unit box (Ignored)
- **subset** (*numpy.array*) – Coordinates to perturb (Ignored)
- **proj_fun** (*Object*) – Routine for projecting infeasible points onto the feasible region
- **merit** (*Object*) – Merit function for selecting candidate points (Ignored)

Returns Points selected for evaluation, of size npts x dim

Return type numpy.array

remove_point (*x*)
Remove x from proposed_points

This removes x from the list of proposed points in the case where the optimization strategy decides to not evaluate x.

Parameters **x** (*numpy.array*) – Point to be removed

Returns True if points was removed, False otherwise

Type bool

class pySOT.adaptive_sampling.**MultiSampling** (*strategy_list*, *cycle*)
Maintains a list of adaptive sampling methods

A collection of adaptive sampling methods and weights so that the user can use multiple adaptive sampling methods for the same optimization problem. This object keeps an internal list of proposed points in order to be able to compute the minimum distance from a point to all proposed evaluations. This list has to be reset each time the optimization algorithm restarts

Parameters

- **strategy_list** (*list*) – List of adaptive sampling methods to use
- **cycle** (*list*) – List of integers that specifies the sampling order, e.g., [0, 0, 1] uses method1, method1, method2, method1, method1, method2, ...

Raises **ValueError** – If cycle is incorrect

Variables

- **sampling_strategies** – List of adaptive sampling methods to use
- **cycle** – List that specifies the sampling order
- **nstrats** – Number of adaptive sampling strategies
- **current_strat** – The next adaptive sampling strategy to be used
- **proposed_points** – List of points proposed to the optimization algorithm
- **data** – Optimization problem object
- **fhat** – Response surface object
- **budget** – Remaining evaluation budget

Note: This object needs to be initialized with the init method. This is done when the initial phase has finished.

Todo

Get rid of the proposed_points object and replace it by something that is controlled by the strategy.

---

init(*start_sample*, *fhat*, *budget*)

Initialize the sampling method after the initial phase

This initializes the list of sampling methods after the initial phase has finished and the experimental design has been evaluated. The user provides the points in the experimental design, the surrogate model, and the remaining evaluation budget.

Parameters

- **start_sample** (*numpy.array*) – Points in the experimental design
- **fhat** (*Object*) – Surrogate model
- **budget** (*int*) – Evaluation budget

make_points(*npts*, *xbest*, *sigma*, *subset=None*, *proj_fun=None*, *merit=<function candidate_merit_weighted_distance>*)

Proposes npts new points to evaluate

Parameters

- **npts** (*int*) – Number of points to select
- **xbest** (*numpy.array*) – Best solution found so far
- **sigma** (*float*) – Current sampling radius w.r.t the unit box
- **subset** (*numpy.array*) – Coordinates to perturb
- **proj_fun** (*Object*) – Routine for projecting infeasible points onto the feasible region
- **merit** (*Object*) – Merit function for selecting candidate points

Returns Points selected for evaluation, of size npts x dim

Return type numpy.array

Todo

Change the merit function from being hard-coded

---

remove_point(*x*)

Remove x from proposed_points

This removes x from the list of proposed points in the case where the optimization strategy decides to not evaluate x.

Parameters **x** (*numpy.array*) – Point to be removed

Returns True if points was removed, False otherwise

Type bool

class pySOT.adaptive_sampling.**MultiStartGradient**(*data*, *method='L-BFGS-B'*, *num_restarts=30*)

A Multi-Start Gradient method for minimizing the surrogate model

A wrapper around the scipy.optimize implementation of box-constrained gradient based minimization.

Parameters

- **data** (*Object*) – Optimization problem object
- **method** (*string*) – Optimization method to use. The options are
  - **L-BFGS-B** Quasi-Newton method of Broyden, Fletcher, Goldfarb, and Shanno (BFGS)
  - **TNC** Truncated Newton algorithm
- **num_restarts** (*int*) – Number of restarts for the multi-start gradient

Raises **ValueError** – If number of candidate points is incorrect or if the weights aren't a list in [0, 1]

Variables

- **data** – Optimization problem object
- **fhat** – Response surface object
- **xrange** – Variable ranges, xup - xlow
- **dtol** – Smallest allowed distance between evaluated points 1e-3 * sqrt(dim)
- **bounds** – n x 2 matrix with lower and upper bound constraints
- **proposed_points** – List of points proposed to the optimization algorithm
- **budget** – Remaining evaluation budget

Note: This object needs to be initialized with the init method. This is done when the initial phase has finished.

Note: SLSQP is supposed to work with bound constraints but for some reason it sometimes violates the constraints anyway.

---

init (*start_sample*, *fhat*, *budget*)
Initialize the sampling method after the initial phase

This initializes the list of sampling methods after the initial phase has finished and the experimental design has been evaluated. The user provides the points in the experimental design, the surrogate model, and the remaining evaluation budget.

Parameters

- **start_sample** (*numpy.array*) – Points in the experimental design
- **fhat** (*Object*) – Surrogate model
- **budget** (*int*) – Evaluation budget

make_points (*npts*, *xbest*, *sigma*, *subset=None*, *proj_fun=None*, *merit=None*)
Proposes npts new points to evaluate

Parameters

- **npts** (*int*) – Number of points to select
- **xbest** (*numpy.array*) – Best solution found so far (Ignored)
- **sigma** (*float*) – Current sampling radius w.r.t the unit box (Ignored)
- **subset** (*numpy.array*) – Coordinates to perturb (Ignored)
- **proj_fun** (*Object*) – Routine for projecting infeasible points onto the feasible region
- **merit** (*Object*) – Merit function for selecting candidate points (Ignored)

Returns Points selected for evaluation, of size npts x dim

Return type numpy.array

remove_point (*x*)
Remove x from proposed_points

This removes x from the list of proposed points in the case where the optimization strategy decides to not evaluate x.

Parameters **x** (*numpy.array*) – Point to be removed

Returns True if points was removed, False otherwise

Type bool

pySOT.ensemble_surrogate module

Module ensemble_surrogate

Author David Eriksson <dme65@cornell.edu>

class pySOT.ensemble_surrogate.**EnsembleSurrogate** (*model_list*, *maxp=100*)
Compute and evaluate an ensemble of interpolants.

Maintains a list of surrogates and decides how to weights them by using Dempster-Shafer theory to assign pignistic probabilities based on statistics computed using LOOCV.

Parameters

- **model_list** (*list*) – List of surrogate models
- **maxp** (*int*) – Maximum number of points

Variables

- **nump** – Current number of points
- **maxp** – Initial maximum number of points (can grow)
- **rhs** – Right hand side for interpolation system
- **x** – Interpolation points
- **fx** – Values at interpolation points
- **dim** – Number of dimensions
- **model_list** – List of surrogate models
- **weights** – Weight for each surrogate model
- **surrogate_list** – List of internal surrogate models for LOOCV

add_point(*xx*, *fx*)
Add a new function evaluation

This function also updates the list of LOOCV surrogate models by cleverly just adding one point to n of the models. The scheme in which new models are built is illustrated below:

2 1 1,2

2,3 1,3 1,2 1,2,3

2,3,4 1,3,4 1,2,4 1,2,3 1,2,3,4

2,3,4,5 1,3,4,5 1,2,4,5 1,2,3,5 1,2,3,4 1,2,3,4,5

Parameters

- **xx** (*numpy.array*) – Point to add
- **fx** (*float*) – The function value of the point to add

compute_weights()
Compute mode weights

Given n observations we use n surrogates built with n-1 of the points in order to predict the value at the removed point. Based on these n predictions we calculate three different statistics:

•Correlation coefficient with true function values

•Root mean square deviation

•Mean absolute error

Based on these three statistics we compute the model weights by applying Dempster-Shafer theory to first compute the pignistic probabilities, which are taken as model weights.

Returns Model weights

Return type numpy.array

deriv(*x*, *d=None*)
Evaluate the derivative of the ensemble surrogate at the point x

Parameters **x** (*numpy.array*) – Point for which we want to compute the RBF gradient

Returns Derivative of the ensemble surrogate at x

Return type numpy.array

eval(*x*, *ds=None*)
Evaluate the ensemble surrogate the point xx

Parameters

- **x** (*numpy.array*) – Point where to evaluate
- **ds** (*None*) – Not used

Returns Value of the ensemble surrogate at x

Return type float

evals(*x*, *ds=None*)
Evaluate the ensemble surrogate at the points xx

Parameters

- **x** (*numpy.array*) – Points where to evaluate, of size npts x dim
- **ds** (*numpy.array*) – Distances between the centers and the points x, of size npts x ncenters

Returns Values of the ensemble surrogate at x, of length npts

Return type numpy.array

get_fx()
Get the list of function values for the data points.

Returns List of function values

Return type numpy.array

get_x()
Get the list of data points

Returns List of data points

Return type numpy.array

reset()
Reset the ensemble surrogate.

pySOT.experimental_design module

Module experimental_design

Author David Eriksson <dme65@cornell.edu> Yi Shen <ys623@cornell.edu>

class pySOT.experimental_design.**BoxBehnken**(*dim*)
Box-Behnken experimental design

The Box-Behnken experimental design consists of the midpoint of the edges plus a center point of the unit hypercube

Parameters **dim** (*int*) – Number of dimensions

Variables

- **dim** – Number of dimensions
- **npts** – Number of desired sampling points (2^dim)

generate_points()
Generate a matrix with the initial sample points, scaled to the unit hypercube

Returns Box-Behnken design in the unit cube of size npts x dim

Return type numpy.array

class pySOT.experimental_design.**LatinHypercube**(*dim*, *npts*, *criterion='c'*)
Latin Hypercube experimental design

Parameters

- **dim** (*int*) – Number of dimensions
- **npts** (*int*) – Number of desired sampling points
- **criterion** (*string*) – Sampling criterion
  - **"center" or "c"** center the points within the sampling intervals
  - **"maximin" or "m"** maximize the minimum distance between points, but place the point in a randomized location within its interval
  - **"centermaximin" or "cm"** same as "maximin", but centered within the intervals
  - **"correlation" or "corr"** minimize the maximum correlation coefficient

Variables

- **dim** – Number of dimensions
- **npts** – Number of desired sampling points
- **criterion** – A string that specifies how to sample

generate_points()
Generate a matrix with the initial sample points, scaled to the unit hypercube

Returns Latin hypercube design in the unit cube of size npts x dim

Return type numpy.array

class pySOT.experimental_design.**SymmetricLatinHypercube**(*dim*, *npts*)
Symmetric Latin Hypercube experimental design

Parameters

- **dim** (*int*) – Number of dimensions
- **npts** (*int*) – Number of desired sampling points

Variables

- **dim** – Number of dimensions
- **npts** – Number of desired sampling points

generate_points()
Generate a matrix with the initial sample points, scaled to the unit hypercube

Returns Symmetric Latin hypercube design in the unit cube of size npts x dim that is of full rank

Return type numpy.array

Raises **ValueError** – Unable to find an SLHD of rank at least dim + 1

class pySOT.experimental_design.**TwoFactorial**(*dim*)
Two-factorial experimental design

The two-factorial experimental design consists of the corners of the unit hypercube, and hence 2^{dim} points.

Parameters **dim** (*int*) – Number of dimensions

Raises **ValueError** – If dim >= 15

Variables

- **dim** – Number of dimensions
- **npts** – Number of desired sampling points (2^dim)

generate_points()
Generate a matrix with the initial sample points, scaled to the unit hypercube

Returns Full-factorial design in the unit cube of size (2^dim) x dim

Return type numpy.array

pySOT.heuristic_methods module

Module heuristic_methods

Author David Eriksson <dme65@cornell.edu>

class pySOT.heuristic_methods.**GeneticAlgorithm**(*function*, *dim*, *xlow*, *xup*, *intvar=None*, *popsize=100*, *ngen=100*, *start='SLHD'*, *proj_fun=None*)
Genetic algorithm

This is an implementation of the real-valued Genetic algorithm that is useful for optimizing on a surrogate model, but it can also be used on its own. The mutations are normally distributed perturbations, the selection mechanism is a tournament selection, and the crossover oepration is the standard linear combination taken at a randomly generated cutting point.

The number of evaluations are popsize x ngen

Parameters

- **function** (*Object*) – Function that can be used to evaluate the entire population. It needs to take an input of size nindividuals x nvariables and return a numpy.array of length nindividuals
- **dim** (*int*) – Number of dimensions
- **xlow** (*numpy.array*) – Lower variable bounds, of length dim
- **xup** (*numpy.array*) – Lower variable bounds, of length dim
- **intvar** (*list*) – List of indices with the integer valued variables (e.g., [0, 1, 5])
- **popsize** (*int*) – Population size
- **ngen** (*int*) – Number of generations
- **start** (*string*) – Method for generating the initial population
- **proj_fun** (*Object*) – Function that can project ONE infeasible individual onto the feasible region

Variables

- **nvariables** – Number of variables (dimensions) of the objective function
- **nindividuals** – population size
- **lower_boundary** – lower bounds for the optimization problem
- **upper_boundary** – upper bounds for the optimization problem
- **integer_variables** – List of variables that are integer valued
- **start** – Method for generating the initial population
- **sigma** – Perturbation radius. Each pertubation is N(0, sigma)
- **p_mutation** – Mutation probability (1/dim)
- **tournament_size** – Size of the tournament (5)
- **p_cross** – Cross-over probability (0.9)
- **ngenerations** – Number of generations
- **function** – Object that can be used to evaluate the objective function
- **projfun** – Function that can be used to project an individual onto the feasible region

optimize()
Method used to run the Genetic algorithm

Returns Returns the best individual and its function value

Return type numpy.array, float

pySOT.gp_regression module

Module gp_regression

Author David Eriksson <dme65@cornell.edu>

class pySOT.gp_regression.**GPRegression**(*maxp=100*, *gp=None*)
Compute and evaluate a GP

Gaussian Process Regression object.

Depends on scitkit-learn==0.18.1.

More details: http://scikit-learn.org/stable/modules/generated/sklearn.gaussian_process.GaussianProcessRegressor.html

Parameters

- **maxp** (*int*) – Initial capacity
- **gp** (*GaussianProcessRegressor*) – GP object (can be None)

Variables

- **nump** – Current number of points
- **maxp** – Initial maximum number of points (can grow)
- **x** – Interpolation points
- **fx** – Function evaluations of interpolation points
- **gp** – Object of type GaussianProcessRegressor

- **dim** – Number of dimensions
- **model** – MARS interpolation model

add_point(*xx*, *fx*)
Add a new function evaluation

Parameters

- **xx** (*numpy.array*) – Point to add
- **fx** (*float*) – The function value of the point to add

deriv(*x*, *ds=None*)
Evaluate the GP regression object at a point x

Parameters

- **x** (*numpy.array*) – Point for which we want to compute the GP regression gradient
- **ds** (*None*) – Not used

Returns Derivative of the GP regression object at x

Return type numpy.array

eval(*x*, *ds=None*)
Evaluate the GP regression object at the point x

Parameters

- **x** (*numpy.array*) – Point where to evaluate
- **ds** (*None*) – Not used

Returns Value of the GP regression obejct at x

Return type float

evals(*x*, *ds=None*)
Evaluate the GP regression object at the points x

Parameters

- **x** (*numpy.array*) – Points where to evaluate, of size npts x dim
- **ds** (*None*) – Not used

Returns Values of the GP regression object at x, of length npts

Return type numpy.array

get_fx()
Get the list of function values for the data points.

Returns List of function values

Return type numpy.array

get_x()
Get the list of data points

Returns List of data points

Return type numpy.array

reset()
Reset the interpolation.

pySOT.mars_interpolant module

Module mars_interpolant

Author Yi Shen <ys623@cornell.edu>

class pySOT.mars_interpolant.**MARSInterpolant**(*maxp=100*)

Compute and evaluate a MARS interpolant

MARS builds a model of the form

$$\hat{f}(x) = \sum_{i=1}^{k} c_i B_i(x).$$

The model is a weighted sum of basis functions $B_i(x)$. Each basis function $B_i(x)$ takes one of the following three forms:

1. a constant 1.
2. a hinge function of the form $\max(0, x - const)$ or $\max(0, const - x)$. MARS automatically selects variables and values of those variables for knots of the hinge functions.
3. a product of two or more hinge functions. These basis functions c an model interaction between two or more variables.

Parameters **maxp** (*int*) – Initial capacity

Variables

- **nump** – Current number of points
- **maxp** – Initial maximum number of points (can grow)
- **x** – Interpolation points
- **fx** – Function evaluations of interpolation points
- **dim** – Number of dimensions
- **model** – MARS interpolation model

add_point(*xx*, *fx*)

Add a new function evaluation

Parameters

- **xx** (*numpy.array*) – Point to add
- **fx** (*float*) – The function value of the point to add

deriv(*x*, *ds=None*)

Evaluate the derivative of the MARS interpolant at a point x

Parameters

- **x** (*numpy.array*) – Point for which we want to compute the MARS gradient
- **ds** (*None*) – Not used

Returns Derivative of the MARS interpolant at x

Return type numpy.array

eval(*x*, *ds=None*)

Evaluate the MARS interpolant at the point x

Parameters

- **x** (*numpy.array*) – Point where to evaluate
- **ds** (*None*) – Not used

Returns Value of the MARS interpolant at x

Return type float

evals(*x*, *ds=None*)
Evaluate the MARS interpolant at the points x

Parameters

- **x** (*numpy.array*) – Points where to evaluate, of size npts x dim
- **ds** (*None*) – Not used

Returns Values of the MARS interpolant at x, of length npts

Return type numpy.array

get_fx()
Get the list of function values for the data points.

Returns List of function values

Return type numpy.array

get_x()
Get the list of data points

Returns List of data points

Return type numpy.array

reset()
Reset the interpolation.

pySOT.merit_functions module

Module merit_functions

Author David Eriksson <dme65@cornell.edu>, David Bindel <bindel@cornell.edu>

pySOT.merit_functions.**candidate_merit_weighted_distance**(*cand*, *npts=1*)
Weighted distance merit function for the candidate points based methods

Parameters

- **cand** (*Object*) – Candidate point object
- **npts** (*int*) – Number of points selected for evaluation

Returns Points selected for evaluation, of size npts x dim

Return type numpy.array

pySOT.poly_regression module

Module poly_regression

Author David Bindel <bindel@cornell.edu>

class pySOT.poly_regression.**PolyRegression**(*bounds*, *basisp*, *maxp=100*)
Compute and evaluate a polynomial regression surface.

Parameters

- **bounds** (*numpy.array*) – a (dims, 2) array of lower and upper bounds in each coordinate
- **basisp** (*numpy.array*) – a (nbasis, dims) array, where the ith basis function is prod_j L_basisp(i,j)(x_j), L_k = the degree k Legendre polynomial
- **maxp** (*int*) – Initial point capacity

Variables

- **nump** – Current number of points
- **maxp** – Initial maximum number of points (can grow)
- **x** – Interpolation points
- **fx** – Function evaluations of interpolation points
- **bounds** – Upper and lower bounds, one row per dimension
- **dim** – Number of dimensions
- **basisp** – Multi-indices representing terms in a tensor poly basis Each row is a list of dim indices indicating a polynomial degree in the associated dimension.
- **updated** – True if the RBF coefficients are up to date

add_point(*xx*, *fx*)
Add a new function evaluation

Parameters

- **xx** – Point to add
- **fx** – The function value of the point to add

deriv(*x*, *ds=None*)
Evaluate the derivative of the regression surface at a point x

Parameters

- **x** (*numpy.array*) – Point where to evaluate
- **ds** (*None*) – Not used

Returns Derivative of the polynomial at x

Return type numpy.array

eval(*x*, *ds=None*)
Evaluate the regression surface at point xx

Parameters

- **x** (*numpy.array*) – Point where to evaluate
- **ds** (*None*) – Not used

Returns Prediction at the point x

Return type float

evals(*x*, *ds=None*)
Evaluate the regression surface at points x

Parameters

- **x** (*numpy.array*) – Points where to evaluate, of size npts x dim
- **ds** (*None*) – Not used

Returns Prediction at the points x

Return type float

get_fx()
Get the list of function values for the data points.

Returns List of function values

Return type numpy.array

get_x()
Get the list of data points

Returns List of data points

Return type numpy.array

reset()
Reset the object.

pySOT.poly_regression.**basis_HC**(*n*, *d*)
Generate list of shape functions for HC poly space.

Parameters

- **n** (*int*) – Dimension of the space
- **d** (*int*) – Degree bound

Returns An N-by-n matrix with S(i,j) = degree of variable j in shape i

Return type numpy.array

pySOT.poly_regression.**basis_SM**(*n*, *d*)
Generate list of shape functions for SM poly space.

Parameters

- **n** (*int*) – Dimension of the space
- **d** (*int*) – Degree bound

Returns An N-by-n matrix with S(i,j) = degree of variable j in shape i

Return type numpy.array

pySOT.poly_regression.**basis_TD**(*n*, *d*)
Generate list of shape functions for TD poly space.

Parameters

- **n** (*int*) – Dimension of the space
- **d** (*int*) – Degree bound

Returns An N-by-n matrix with S(i,j) = degree of variable j in shape i

Return type numpy.array

pySOT.poly_regression.**basis_TP**(*n*, *d*)
Generate list of shape functions for TP poly space.

Parameters

- **n** (*int*) – Dimension of the space
- **d** (*int*) – Degree bound

Returns An N-by-n matrix with S(i,j) = degree of variable j in shape i There are N = n^d shapes.

Return type numpy.array

pySOT.poly_regression.**basis_base**(*n*, *testf*)
Generate list of shape functions for a subset of a TP poly space.

Parameters

- **n** (*int*) – Dimension of the space
- **testf** (*Object*) – Return True if a given multi-index is in range

Returns An N-by-n matrix with S(i,j) = degree of variable j in shape i

Return type numpy.array

pySOT.poly_regression.**dlegendre**(*x*, *d*)
Evaluate Legendre polynomial derivatives at all coordinates in x.

Parameters

- **x** (*numpy.array*) – Array of coordinates
- **d** (*int*) – Max degree of polynomials

Returns x.shape-by-d arrays of Legendre polynomial values and derivatives

Return type numpy.array

pySOT.poly_regression.**legendre**(*x*, *d*)
Evaluate Legendre polynomials at all coordinates in x.

Parameters

- **x** (*numpy.array*) – Array of coordinates
- **d** (*int*) – Max degree of polynomials

Returns A x.shape-by-d array of Legendre polynomial values

Return type numpy.array

pySOT.poly_regression.**test_legendre1**()

pySOT.poly_regression.**test_legendre2**()

pySOT.poly_regression.**test_poly**()

pySOT.kernels module

Module kernels

Author David Eriksson <dme65@cornell.edu>,

class pySOT.kernels.**CubicKernel**
Cubic RBF kernel

This is a basic class for the Cubic RBF kernel: $\varphi(r) = r^3$ which is conditionally positive definite of order 2.

deriv(*dists*)
evaluates the derivative of the Cubic kernel for a distance matrix

Parameters **dists** (*numpy.array*) – Distance input matrix

Returns a matrix where element (i, j) is $3\|x_i - x_j\|^2$

Return type numpy.array

eval(*dists*)
evaluates the Cubic kernel for a distance matrix

Parameters **dists** (*numpy.array*) – Distance input matrix

Returns a matrix where element (i, j) is $\|x_i - x_j\|^3$

Return type numpy.array

order()
returns the order of the Cubic RBF kernel

Returns 2

Return type int

phi_zero()
returns the value of $\varphi(0)$ for Cubic RBF kernel

Returns 0

Return type float

class pySOT.kernels.**LinearKernel**
Linear RBF kernel

This is a basic class for the Linear RBF kernel: $\varphi(r) = r$ which is conditionally positive definite of order 1.

deriv(*dists*)
evaluates the derivative of the Cubic kernel for a distance matrix

Parameters **dists** (*numpy.array*) – Distance input matrix

Returns a matrix where element (i, j) is 1

Return type numpy.array

eval(*dists*)
evaluates the Linear kernel for a distance matrix

Parameters **dists** (*numpy.array*) – Distance input matrix

Returns a matrix where element (i, j) is $\|x_i - x_j\|$

Return type numpy.array

order()
returns the order of the Linear RBF kernel

Returns 1

Return type int

phi_zero()
returns the value of $\varphi(0)$ for Linear RBF kernel

Returns 0

Return type float

class pySOT.kernels.**TPSKernel**
Thin-plate spline RBF kernel

This is a basic class for the TPS RBF kernel: $\varphi(r) = r^2 \log(r)$ which is conditionally positive definite of order 2.

deriv(*dists*)
evaluates the derivative of the Cubic kernel for a distance matrix

Parameters **dists** (*numpy.array*) – Distance input matrix

Returns a matrix where element (i, j) is $\|x_i - x_j\|(1 + 2\log(\|x_i - x_j\|))$

Return type numpy.array

eval(*dists*)
evaluates the Cubic kernel for a distance matrix

Parameters **dists** (*numpy.array*) – Distance input matrix

Returns a matrix where element (i, j) is $\|x_i - x_j\|^2 \log(\|x_i - x_j\|)$

Return type numpy.array

order()
returns the order of the TPS RBF kernel

Returns 2

Return type int

phi_zero()
returns the value of $\varphi(0)$ for TPS RBF kernel

Returns 0

Return type float

pySOT.tails module

Module tails

Author David Eriksson <dme65@cornell.edu>,

class pySOT.tails.**ConstantTail**
Constant polynomial tail

This is a standard linear polynomial in d-dimension, built from the basis $\{1\}$.

degree()
returns the degree of the constant polynomial tail

Returns 0

Return type int

deriv(*x*)
evaluates the gradient of the linear polynomial tail for one point

Parameters **x** (*numpy.array*) – Point to evaluate, of length dim

Returns A numpy.array of size dim x dim_tail(dim)

Return type numpy.array

dim_tail(*dim*)
returns the dimensionality of the constant polynomial space for a given dimension

Parameters **dim** (*int*) – Number of dimensions of the Cartesian space

Returns 1

Return type int

eval(*X*)
evaluates the constant polynomial tail for a set of points

Parameters **X** (*numpy.array*) – Points to evaluate, of size npts x dim

Returns A numpy.array of size npts x dim_tail(dim)

Return type numpy.array

class pySOT.tails.**LinearTail**
Linear polynomial tail

This is a standard linear polynomial in d-dimension, built from the basis $\{1, x_1, x_2, \ldots, x_d\}$.

degree()
returns the degree of the linear polynomial tail

Returns 1

Return type int

deriv(*x*)
evaluates the gradient of the linear polynomial tail for one point

Parameters **x** (*numpy.array*) – Point to evaluate, of length dim

Returns A numpy.array of size dim x dim_tail(dim)

Return type numpy.array

dim_tail(*dim*)
returns the dimensionality of the linear polynomial space for a given dimension

Parameters **dim** (*int*) – Number of dimensions of the Cartesian space

Returns 1 + dim

Return type int

eval(*X*)
evaluates the linear polynomial tail for a set of points

Parameters **X** (*numpy.array*) – Points to evaluate, of size npts x dim

Returns A numpy.array of size npts x dim_tail(dim)

Return type numpy.array

pySOT.rbf module

Module rbf

Author David Eriksson <dme65@cornell.edu>, David Bindel <bindel@cornell.edu>

class pySOT.rbf.**RBFInterpolant**(*kernel=<class 'pySOT.kernels.CubicKernel'>*, *tail=<class 'pySOT.tails.LinearTail'>*, *maxp=500*, *eta=1e-08*)

Compute and evaluate RBF interpolant.

Manages an expansion of the form

$$f(x) = \sum_j c_j \phi(\|x - x_j\|) + \sum_j \lambda_j p_j(x)$$

where the functions $p_j(x)$ are low-degree polynomials. The fitting equations are

$$\begin{bmatrix} \eta I & P^T \\ P & \Phi + \eta I \end{bmatrix} \begin{bmatrix} \lambda \\ c \end{bmatrix} = \begin{bmatrix} 0 \\ f \end{bmatrix}$$

where $P_{ij} = p_j(x_i)$ and $\Phi_{ij} = \phi(\|x_i - x_j\|)$. The regularization parameter η allows us to avoid problems with potential poor conditioning of the system. The regularization parameter can either be fixed or estimated via LOOCV. Specify eta='adapt' for estimation.

Parameters

- **kernel** (*Kernel*) – RBF kernel object
- **tail** (*Tail*) – RBF polynomial tail object
- **maxp** (*int*) – Initial point capacity
- **eta** (*float or 'adapt'*) – Regularization parameter

Variables

- **kernel** – RBF kernel
- **tail** – RBF tail
- **eta** – Regularization parameter
- **ntail** – Number of tail functions
- **nump** – Current number of points
- **maxp** – Initial maximum number of points (can grow)
- **A** – Interpolation system matrix
- **LU** – LU-factorization of the RBF system
- **piv** – pivot vector for the LU-factorization
- **rhs** – Right hand side for interpolation system
- **x** – Interpolation points
- **fx** – Values at interpolation points
- **c** – Expansion coefficients
- **dim** – Number of dimensions
- **ntail** – Number of tail functions
- **updated** – True if the RBF coefficients are up to date

add_point(*xx*, *fx*)
Add a new function evaluation

Parameters

- **xx** (*numpy.array*) – Point to add
- **fx** (*float*) – The function value of the point to add

coeffs()
Compute the expansion coefficients

Returns Expansion coefficients

Return type numpy.array

deriv(*x*, *ds=None*)
Evaluate the derivative of the RBF interpolant at a point x

Parameters

- **x** (*numpy.array*) – Point for which we want to compute the RBF gradient
- **ds** (*numpy.array*) – Distances between the centers and the point x

Returns Derivative of the RBF interpolant at x

Return type numpy.array

eval(*x*, *ds=None*)
Evaluate the RBF interpolant at the point x

Parameters **x** (*numpy.array*) – Point where to evaluate

Returns Value of the RBF interpolant at x

Return type float

evals(*x*, *ds=None*)
Evaluate the RBF interpolant at the points x

Parameters

- **x** (*numpy.array*) – Points where to evaluate, of size npts x dim
- **ds** (*numpy.array*) – Distances between the centers and the points x, of size npts x ncenters

Returns Values of the rbf interpolant at x, of length npts

Return type numpy.array

get_fx()
Get the list of function values for the data points.

Returns List of function values

Return type numpy.array

get_x()
Get the list of data points

Returns List of data points

Return type numpy.array

reset()
Reset the RBF interpolant

transform_fx(*fx*)
Replace f with transformed function values for the fitting

Parameters **fx** (*numpy.array*) – Transformed function values

pySOT.rs_wrappers module

Module rs_wrappers

Author David Bindel <bindel@cornell.edu>

class pySOT.rs_wrappers.**RSCapped**(*model*, *transformation=None*)
Cap adapter for response surfaces.

This adapter takes an existing response surface and replaces it with a modified version in which the function values are replaced according to some transformation. A very common transformation is to replace all values above the median by the median in order to reduce the influence of large function values.

Parameters

- **model** (*Object*) – Original response surface object
- **transformation** (*Object*) – Function value transformation object. Median capping is used if no object (or None) is provided

Variables

- **transformation** – Object used to transform the function values.
- **model** – original response surface object
- **fvalues** – Function values
- **nump** – Current number of points
- **maxp** – Initial maximum number of points (can grow)
- **updated** – True if the surface is updated

add_point(*xx*, *fx*)
Add a new function evaluation

Parameters

- **xx** (*numpy.array*) – Point to add
- **fx** (*float*) – The function value of the point to add

deriv(*x*, *ds=None*)
Evaluate the derivative of the capped interpolant at a point x

Parameters

- **x** (*numpy.array*) – Point for which we want to compute the RBF gradient
- **ds** (*numpy.array*) – Distances between the centers and the point x

Returns Derivative of the capped interpolant at x

Return type numpy.array

eval(*x*, *ds=None*)
Evaluate the capped interpolant at the point x

Parameters **x** (*numpy.array*) – Point where to evaluate

Returns Value of the RBF interpolant at x

Return type float

evals(*x*, *ds=None*)
Evaluate the capped interpolant at the points x

Parameters

- **x** (*numpy.array*) – Points where to evaluate, of size npts x dim
- **ds** (*numpy.array*) – Distances between the centers and the points x, of size npts x ncenters

Returns Values of the capped interpolant at x, of length npts

Return type numpy.array

get_fx()
Get the list of function values for the data points.

Returns List of function values

Return type numpy.array

get_x()
Get the list of data points

Returns List of data points

Return type numpy.array

reset()
Reset the capped response surface

class pySOT.rs_wrappers.**RSPenalty**(*model*, *evals*, *derivs*)
Penalty adapter for response surfaces.

This adapter can be used for approximating an objective function plus a penalty function. The response surface is fitted only to the objective function and the penalty is added on after.

Parameters

- **model** (*Object*) – Original response surface object
- **evals** (*Object*) – Object that takes the response surface and the points and adds up the response surface value and the penalty function value
- **devals** (*Object*) – Object that takes the response surface and the points and adds up the response surface derivative and the penalty function derivative

Variables

- **eval_method** – Object that takes the response surface and the points and adds up the response surface value and the penalty function value
- **deval_method** – Object that takes the response surface and the points and adds up the response surface derivative and the penalty function derivative
- **model** – original response surface object
- **fvalues** – Function values
- **nump** – Current number of points
- **maxp** – Initial maximum number of points (can grow)
- **updated** – True if the surface is updated

add_point(*xx*, *fx*)
Add a new function evaluation

Parameters

- **xx** (*numpy.array*) – Point to add
- **fx** (*float*) – The function value of the point to add

deriv(*x*, *ds=None*)
Evaluate the derivative of the penalty adapter at x

Parameters

- **x** (*numpy.array*) – Point for which we want to compute the gradient
- **ds** (*None*) – Not used

Returns Derivative of the interpolant at x

Return type numpy.array

eval(*x*, *ds=None*)
Evaluate the penalty adapter interpolant at the point xx

Parameters

- **x** (*numpy.array*) – Point where to evaluate
- **ds** (*None*) – Not used

Returns Value of the interpolant at x

Return type float

evals(*x*, *ds=None*)
Evaluate the penalty adapter at the points xx

Parameters

- **x** (*numpy.array*) – Points where to evaluate, of size npts x dim
- **ds** (*None*) – Not used

Returns Values of the interpolant at x, of length npts

Return type numpy.array

get_fx()
Get the list of function values for the data points.

Returns List of function values

Return type numpy.array

get_x()
Get the list of data points

Returns List of data points

Return type numpy.array

reset()
Reset the capped response surface

class pySOT.rs_wrappers.**RSUnitbox**(*model*, *data*)
Unit box adapter for response surfaces

This adapter takes an existing response surface and replaces it with a modified version where the domain is rescaled to the unit box. This is useful for response surfaces that are sensitive to scaling, such as radial basis functions.

Parameters

- **model** (*Object*) – Original response surface object
- **data** (*Object*) – Optimization problem object

Variables

- **data** – Optimization problem object
- **model** – original response surface object
- **fvalues** – Function values
- **nump** – Current number of points
- **maxp** – Initial maximum number of points (can grow)
- **updated** – True if the surface is updated

add_point (*xx*, *fx*)
Add a new function evaluation

Parameters

- **xx** (*numpy.array*) – Point to add
- **fx** (*float*) – The function value of the point to add

deriv (*x*, *ds=None*)
Evaluate the derivative of the rbf interpolant at x

Parameters

- **x** (*numpy.array*) – Point for which we want to compute the MARS gradient
- **ds** (*None*) – Not used

Returns Derivative of the MARS interpolant at x

Return type numpy.array

eval (*x*, *ds=None*)
Evaluate the response surface at the point xx

Parameters

- **x** (*numpy.array*) – Point where to evaluate
- **ds** (*None*) – Not used

Returns Value of the interpolant at x

Return type float

evals (*x*, *ds=None*)
Evaluate the capped rbf interpolant at the points xx

Parameters

- **x** (*numpy.array*) – Points where to evaluate, of size npts x dim
- **ds** (*None*) – Not used

Returns Values of the MARS interpolant at x, of length npts

Return type numpy.array

get_fx()
Get the list of function values for the data points.

Returns List of function values

Return type numpy.array

get_x()
Get the list of data points

Returns List of data points

Return type numpy.array

reset()
Reset the capped response surface

pySOT.sot_sync_strategies module

Module sot_sync_strategies

Author David Bindel <bindel@cornell.edu>, David Eriksson <dme65@cornell.edu>

class pySOT.sot_sync_strategies.**SyncStrategyNoConstraints**(*worker_id*, *data*, *response_surface*, *maxeval*, *nsamples*, *exp_design=None*, *sampling_method=None*, *extra=None*, *extra_vals=None*)
Parallel synchronous optimization strategy without non-bound constraints.

This class implements the parallel synchronous SRBF strategy described by Regis and Shoemaker. After the initial experimental design (which is embarrassingly parallel), the optimization proceeds in phases. During each phase, we allow nsamples simultaneous function evaluations. We insist that these evaluations run to completion – if one fails for whatever reason, we will resubmit it. Samples are drawn randomly from around the current best point, and are sorted according to a merit function based on distance to other sample points and predicted function values according to the response surface. After several successive significant improvements, we increase the sampling radius; after several failures to improve the function value, we decrease the sampling radius. We restart once the sampling radius decreases below a threshold.

Parameters

- **worker_id** (*int*) – Start ID in a multi-start setting
- **data** (*Object*) – Problem parameter data structure
- **response_surface** (*Object*) – Surrogate model object
- **maxeval** (*int*) – Stopping criterion. If positive, this is an evaluation budget. If negative, this is a time budget in seconds.
- **nsamples** (*int*) – Number of simultaneous fevals allowed
- **exp_design** (*Object*) – Experimental design
- **sampling_method** (*Object*) – Sampling method for finding points to evaluate
- **extra** (*numpy.array*) – Points to be added to the experimental design

- **extra_vals** (*numpy.array*) – Values of the points in extra (if known). Use nan for values that are not known.

adjust_step()
Adjust the sampling radius sigma.

After succtol successful steps, we cut the sampling radius; after failtol failed steps, we double the sampling radius.

check_common()
Checks that the inputs are correct

check_input()
Checks that the inputs are correct

log_completion(*record*)
Record a completed evaluation to the log.

Parameters **record** (*Object*) – Record of the function evaluation

on_complete(*record*)
Handle completed function evaluation.

When a function evaluation is completed we need to ask the constraint handler if the function value should be modified which is the case for say a penalty method. We also need to print the information to the logfile, update the best value found so far and notify the GUI that an evaluation has completed.

Parameters **record** (*Object*) – Evaluation record

on_reply_accept(*proposal*)

proj_fun(*x*)
Projects a set of points onto the feasible region

Parameters **x** (*numpy.array*) – Points, of size npts x dim

Returns Projected points

Return type numpy.array

propose_action()
Propose an action

sample_adapt()
Generate and queue samples from the search strategy

sample_initial()
Generate and queue an initial experimental design.

start_batch()
Generate and queue a new batch of points

class pySOT.sot_sync_strategies.**SyncStrategyPenalty**(*worker_id*, *data*, *response_surface*, *maxeval*, *nsamples*, *exp_design=None*, *sampling_method=None*, *extra=None*, *penalty=1000000.0*)
Parallel synchronous optimization strategy with non-bound constraints.

This is an extension of SyncStrategyNoConstraints that also works with bound constraints. We currently only allow inequality constraints, since the candidate based methods don't work well with equality constraints. We also assume that the constraints are cheap to evaluate, i.e., so that it is easy to check if a given point is feasible. More strategies that can handle expensive constraints will be added.

We use a penalty method in the sense that we try to minimize:

$$f(x) + \mu \sum_j (\max(0, g_j(x))^2$$

where $g_j(x) \leq 0$ are cheap inequality constraints. As a measure of promising function values we let all infeasible points have the value of the feasible candidate point with the worst function value, since large penalties makes it impossible to distinguish between feasible points.

When it comes to the value of μ, just choose a very large value.

Parameters

- **worker_id** (*int*) – Start ID in a multi-start setting
- **data** (*Object*) – Problem parameter data structure
- **response_surface** (*Object*) – Surrogate model object
- **maxeval** (*int*) – Function evaluation budget
- **nsamples** (*int*) – Number of simultaneous fevals allowed
- **exp_design** (*Object*) – Experimental design
- **sampling_method** (*Object*) – Sampling method for finding points to evaluate
- **extra** (*numpy.array*) – Points to be added to the experimental design
- **penalty** (*float*) – Penalty for violating constraints

check_input()
Checks that the inputs are correct

on_complete(*record*)
Handle completed function evaluation.

When a function evaluation is completed we need to ask the constraint handler if the function value should be modified which is the case for say a penalty method. We also need to print the information to the logfile, update the best value found so far and notify the GUI that an evaluation has completed.

Parameters **record** (*Object*) – Evaluation record

penalty_fun(*xx*)
Computes the penalty for constraint violation

Parameters **xx** (*numpy.array*) – Points to compute the penalty for

Returns Penalty for constraint violations

Return type numpy.array

class pySOT.sot_sync_strategies.**SyncStrategyProjection**(*worker_id*, *data*, *response_surface*, *maxeval*, *nsamples*, *exp_design=None*, *sampling_method=None*, *extra=None*, *proj_fun=None*)
Parallel synchronous optimization strategy with non-bound constraints. It uses a supplied method to project proposed points onto the feasible region in order to always evaluate feasible points which is useful in situations where it is easy to project onto the feasible region and where the objective function is nonsensical for infeasible points.

This is an extension of SyncStrategyNoConstraints that also works with bound constraints.

Parameters

- **worker_id** (*int*) – Start ID in a multi-start setting
- **data** (*Object*) – Problem parameter data structure
- **response_surface** (*Object*) – Surrogate model object
- **maxeval** (*int*) – Function evaluation budget
- **nsamples** (*int*) – Number of simultaneous fevals allowed
- **exp_design** (*Object*) – Experimental design
- **sampling_method** (*Object*) – Sampling method for finding points to evaluate
- **extra** (*numpy.array*) – Points to be added to the experimental design
- **proj_fun** (*Object*) – Function that projects one point onto the feasible region

check_input()
Checks that the inputs are correct

proj_fun(*x*)
Projects a set of points onto the feasible region

Parameters **x** (*numpy.array*) – Points, of size npts x dim

Returns Projected points

Return type numpy.array

pySOT.test_problems module

Module test_problems

Author David Eriksson <dme65@cornell.edu>, David Bindel <bindel@cornell.edu>

class pySOT.test_problems.**Ackley**(*dim=10*)
Ackley function

$$f(x_1, \ldots, x_n) = -20 \exp\left(-0.2\sqrt{\frac{1}{n}\sum_{j=1}^{n} x_j^2}\right) - \exp\left(\frac{1}{n}\sum j = 1^n \cos(2\pi x_j)\right) + 20 - e$$

subject to

$$-15 \leq x_i \leq 20$$

Global optimum: $f(0, 0, ..., 0) = 0$

Parameters **dim** (*int*) – Number of dimensions

Variables

- **dim** – Number of dimensions
- **xlow** – Lower bound constraints
- **xup** – Upper bound constraints
- **info** – Problem information:
- **min** – Global optimum
- **integer** – Integer variables

- **continuous** – Continuous variables

objfunction(*x*)
Evaluate the Ackley function at x

Parameters **x** (*numpy.array*) – Data point

Returns Value at x

Return type float

class pySOT.test_problems.**Exponential**(*dim=10*)
Exponential function

$$f(x_1, \ldots, x_n) = \sum_{j=1}^{n} e^{jx_j} - \sum_{j=1}^{n} e^{-5.12j}$$

subject to

$$-5.12 \leq x_i \leq 5.12$$

Global optimum: $f(0, 0, ..., 0) = 0$

Parameters **dim** (*int*) – Number of dimensions

Variables

- **dim** – Number of dimensions
- **xlow** – Lower bound constraints
- **xup** – Upper bound constraints
- **info** – Problem information
- **min** – Global optimum
- **integer** – Integer variables
- **continuous** – Continuous variables

objfunction(*x*)
Evaluate the Exponential function at x

Parameters **x** (*numpy.array*) – Data point

Returns Value at x

Return type float

class pySOT.test_problems.**Griewank**(*dim=10*)
Griewank function

$$f(x_1, \ldots, x_n) = 1 + \frac{1}{4000} \sum_{j=1}^{n} x_j^2 - \prod_{j=1}^{n} \cos\left(\frac{x_i}{\sqrt{i}}\right)$$

subject to

$$-512 \leq x_i \leq 512$$

Global optimum: $f(0, 0, ..., 0) = 0$

Parameters **dim** (*int*) – Number of dimensions

Variables

- **dim** – Number of dimensions
- **xlow** – Lower bound constraints
- **xup** – Upper bound constraints
- **info** – Problem information
- **min** – Global optimum
- **integer** – Integer variables
- **continuous** – Continuous variables

objfunction(*x*)
Evaluate the Griewank function at x

Parameters **x** (*numpy.array*) – Data point

Returns Value at x

Return type float

class pySOT.test_problems.**Hartman3**(*dim=3*)
Hartman 3 function

Details: http://www.sfu.ca/~ssurjano/hart3.html

Global optimum: $f(0.114614, 0.555649, 0.852547) = -3.86278$

Parameters **dim** (*int*) – Number of dimensions (has to be = 3)

Variables

- **dim** – Number of dimensions
- **xlow** – Lower bound constraints
- **xup** – Upper bound constraints
- **info** – Problem information
- **min** – Global optimum
- **integer** – Integer variables
- **continuous** – Continuous variables

objfunction(*x*)
Evaluate the Hartman 3 function at x

Parameters **x** – Data point

Returns Value at x

class pySOT.test_problems.**Hartman6**(*dim=6*)
Hartman 6 function

Details: http://www.sfu.ca/~ssurjano/hart6.html

Global optimum: $f(0.20169, 0.150011, 0.476874, 0.275332, 0.311652, 0.6573) = -3.32237$

Parameters **dim** (*int*) – Number of dimensions (has to be = 6)

Variables

- **dim** – Number of dimensions
- **xlow** – Lower bound constraints

- **xup** – Upper bound constraints
- **info** – Problem information
- **min** – Global optimum
- **integer** – Integer variables
- **continuous** – Continuous variables

objfunction(*x*)
Evaluate the Hartman 3 function at x

Parameters **x** – Data point

Returns Value at x

class pySOT.test_problems.**Keane**(*dim=10*)
Keane's "bump" function

$$f(x_1, \ldots, x_n) = -\left| \frac{\sum_{j=1}^{n} \cos^4(x_j) - 2\prod_{j=1}^{n} \cos^2(x_j)}{\sqrt{\sum_{j=1}^{n} j x_j^2}} \right|$$

subject to

$$0 \leq x_i \leq 5$$

$$0.75 - \prod_{j=1}^{n} x_j < 0$$

$$\sum_{j=1}^{n} x_j - 7.5n < 0$$

Global optimum: -0.835 for large n

Parameters **dim** (*int*) – Number of dimensions

Variables

- **dim** – Number of dimensions
- **xlow** – Lower bound constraints
- **xup** – Upper bound constraints
- **info** – Problem information
- **min** – Global optimum
- **integer** – Integer variables
- **continuous** – Continuous variables

deriv_ineq_constraints(*x*)
Evaluate the derivative of the Keane inequality constraints at x

Parameters **x** (*numpy.array*) – Data points, of size npts x dim

Returns Derivative at the constraints, of size npts x nconstraints x ndims

Return type float

eval_ineq_constraints(*x*)
Evaluate the Keane inequality constraints at x

Parameters **x** (*numpy.array*) – Data points, of size npts x dim

Returns Value at the constraints, of size npts x nconstraints

Return type float

objfunction(*x*)
Evaluate the Keane function at a point x

Parameters **x** (*numpy.array*) – Data point

Returns Value at x

Return type float

class pySOT.test_problems.**Levy**(*dim=10*)
Ackley function

Details: https://www.sfu.ca/~ssurjano/levy.html

Global optimum: $f(1, 1, ..., 1) = 0$

Parameters **dim** (*int*) – Number of dimensions

Variables

- **dim** – Number of dimensions
- **xlow** – Lower bound constraints
- **xup** – Upper bound constraints
- **info** – Problem information
- **min** – Global optimum
- **integer** – Integer variables
- **continuous** – Continuous variables

objfunction(*x*)
Evaluate the Levy function at x

Parameters **x** – Data point

Returns Value at x

class pySOT.test_problems.**LinearMI**(*dim=5*)
This is a linear mixed integer problem with non-bound constraints

There are 5 variables, the first 3 are discrete and the last 2 are continuous.

Global optimum: $f(1, 0, 0, 0, 0) = -1$

Parameters **dim** (*int*) – Number of dimensions (has to be 5)

Variables

- **dim** – Number of dimensions
- **xlow** – Lower bound constraints
- **xup** – Upper bound constraints
- **info** – Problem information
- **min** – Global optimum
- **integer** – Integer variables

- **continuous** – Continuous variables

eval_ineq_constraints(*x*)
Evaluate the LinearMI inequality constraints at x

Parameters **x** (*numpy.array*) – Data points, of size npts x dim

Returns Value at the constraints, of size npts x nconstraints

Return type float

objfunction(*x*)
Evaluate the LinearMI function at x

Parameters **x** (*numpy.array*) – Data point

Returns Value at x

Return type float

class pySOT.test_problems.**Michalewicz**(*dim=10*)
Michalewicz function

$$f(x_1, \ldots, x_n) = -\sum_{i=1}^{n} \sin(x_i) \sin^{20}\left(\frac{ix_i^2}{\pi}\right)$$

subject to

$$0 \leq x_i \leq \pi$$

Parameters **dim** (*int*) – Number of dimensions

Variables

- **dim** – Number of dimensions
- **xlow** – Lower bound constraints
- **xup** – Upper bound constraints
- **info** – Problem information
- **min** – Global optimum
- **integer** – Integer variables
- **continuous** – Continuous variables

objfunction(*x*)
Evaluate the Michalewicz function at x

Parameters **x** (*numpy.array*) – Data point

Returns Value at x

Return type float

class pySOT.test_problems.**Quartic**(*dim=10*)
Quartic function

$$f(x_1, \ldots, x_n) = \sum_{j=1}^{n} jx_j^4 + random[0, 1)$$

subject to

$$-1.28 \leq x_i \leq 1.28$$

Global optimum: $f(0, 0, ..., 0) = 0 + noise$

Parameters **dim** (*int*) – Number of dimensions

Variables

- **dim** – Number of dimensions
- **xlow** – Lower bound constraints
- **xup** – Upper bound constraints
- **info** – Problem information
- **min** – Global optimum
- **integer** – Integer variables
- **continuous** – Continuous variables

objfunction(*x*)
Evaluate the Quartic function at x

Parameters **x** (*numpy.array*) – Data point

Returns Value at x

Return type float

class pySOT.test_problems.**Rastrigin**(*dim=10*)
Rastrigin function

$$f(x_1, \ldots, x_n) = 10n - \sum_{i=1}^{n} (x_i^2 - 10\cos(2\pi x_i))$$

subject to

$$-5.12 \leq x_i \leq 5.12$$

Global optimum: $f(0, 0, ..., 0) = 0$

Parameters **dim** (*int*) – Number of dimensions

Variables

- **dim** – Number of dimensions
- **xlow** – Lower bound constraints
- **xup** – Upper bound constraints
- **info** – Problem information
- **min** – Global optimum
- **integer** – Integer variables
- **continuous** – Continuous variables

objfunction(*x*)
Evaluate the Rastrigin function at x

Parameters **x** (*numpy.array*) – Data point

Returns Value at x

Return type float

class pySOT.test_problems.**Rosenbrock**(*dim=10*)

Rosenbrock function

$$f(x_1, \ldots, x_n) = \sum_{j=1}^{n-1} \left(100(x_j^2 - x_{j+1})^2 + (1 - x_j)^2\right)$$

subject to

$$-2.048 \leq x_i \leq 2.048$$

Global optimum: $f(1, 1, ..., 1) = 0$

Parameters **dim** (*int*) – Number of dimensions

Variables

- **dim** – Number of dimensions
- **xlow** – Lower bound constraints
- **xup** – Upper bound constraints
- **info** – Problem information
- **min** – Global optimum
- **integer** – Integer variables
- **continuous** – Continuous variables

objfunction(*x*)

Evaluate the Rosenbrock function at x

Parameters **x** (*numpy.array*) – Data point

Returns Value at x

Return type float

class pySOT.test_problems.**SchafferF7**(*dim=10*)

SchafferF7 function

$$f(x_1, \ldots, x_n) = \left[\frac{1}{n-1}\sqrt{s_i} \cdot (\sin(50.0 s_i^{\frac{1}{5}}) + 1)\right]^2$$

where

$$s_i = \sqrt{x_i^2 + x_{i+1}^2}$$

subject to

$$-100 \leq x_i \leq 100$$

Global optimum: $f(0, 0, ..., 0) = 0$

Parameters **dim** (*int*) – Number of dimensions

Variables

- **dim** – Number of dimensions
- **xlow** – Lower bound constraints
- **xup** – Upper bound constraints

- **info** – Problem information
- **min** – Global optimum
- **integer** – Integer variables
- **continuous** – Continuous variables

objfunction(*x*)
Evaluate the SchafferF7 function at x

Parameters **x** (*numpy.array*) – Data point

Returns Value at x

Return type float

class pySOT.test_problems.**Schwefel**(*dim=10*)
Schwefel function

$$f(x_1, \ldots, x_n) = \sum_{j=1}^{n} \left(-x_j \sin(\sqrt{|x_j|}) \right) + 418.982997n$$

subject to

$$-512 \leq x_i \leq 512$$

Global optimum: $f(420.968746, 420.968746, ..., 420.968746) = 0$

Parameters **dim** (*int*) – Number of dimensions

Variables

- **dim** – Number of dimensions
- **xlow** – Lower bound constraints
- **xup** – Upper bound constraints
- **info** – Problem information
- **min** – Global optimum
- **integer** – Integer variables
- **continuous** – Continuous variables

objfunction(*x*)
Evaluate the Schwefel function at x

Parameters **x** (*numpy.array*) – Data point

Returns Value at x

Return type float

class pySOT.test_problems.**Sphere**(*dim=10*)
Sphere function

$$f(x_1, \ldots, x_n) = \sum_{j=1}^{n} x_j^2$$

subject to

$$-5.12 \leq x_i \leq 5.12$$

Global optimum: $f(0, 0, ..., 0) = 0$

Parameters **dim** (*int*) – Number of dimensions

Variables

- **dim** – Number of dimensions
- **xlow** – Lower bound constraints
- **xup** – Upper bound constraints
- **info** – Problem information
- **min** – Global optimum
- **integer** – Integer variables
- **continuous** – Continuous variables

objfunction(*x*)
Evaluate the Sphere function at x

Parameters **x** (*numpy.array*) – Data point

Returns Value at x

Return type float

class pySOT.test_problems.**StyblinskiTang**(*dim=10*)
StyblinskiTang function

$$f(x_1, \ldots, x_n) = \frac{1}{2} \sum_{j=1}^{n} \left(x_j^4 - 16x_j^2 + 5x_j \right)$$

subject to

$$-5 \leq x_i \leq 5$$

Global optimum: $f(-2.903534, -2.903534, ..., -2.903534) = -39.16599 \cdot n$

Parameters **dim** (*int*) – Number of dimensions

Variables

- **dim** – Number of dimensions
- **xlow** – Lower bound constraints
- **xup** – Upper bound constraints
- **info** – Problem information
- **min** – Global optimum
- **integer** – Integer variables
- **continuous** – Continuous variables

objfunction(*x*)
Evaluate the StyblinskiTang function at x

Parameters **x** (*numpy.array*) – Data point

Returns Value at x

Return type float

class pySOT.test_problems.**Whitley**(*dim=10*)
Quartic function

$$f(x_1, \ldots, x_n) = \sum_{i=1}^{n} \sum_{j=1}^{n} \left(\frac{(100(x_i^2 - x_j)^2 + (1 - x_j)^2)^2}{4000} - \cos(100(x_i^2 - x_j)^2 + (1 - x_j)^2) + 1 \right)$$

subject to

$$-10.24 \leq x_i \leq 10.24$$

Global optimum: $f(1, 1, ..., 1) = 0$

Parameters **dim** (*int*) – Number of dimensions

Variables

- **dim** – Number of dimensions
- **xlow** – Lower bound constraints
- **xup** – Upper bound constraints
- **info** – Problem information
- **min** – Global optimum
- **integer** – Integer variables
- **continuous** – Continuous variables

objfunction(*x*)
Evaluate the Whitley function at x

Parameters **x** (*numpy.array*) – Data point

Returns Value at x

Return type float

pySOT.utils module

Module utils

Author David Eriksson <dme65@cornell.edu>

pySOT.utils.**check_opt_prob**(*obj*)
Routine for checking that an implementation of the optimization problem follows the standard. This method checks everything, but can't make sure that the objective function and constraint methods return values of the correct type since this would involve actually evaluating the objective function which isn't feasible when the evaluations are expensive. If some test fails, an exception is raised through assert.

Parameters **obj** (*Object*) – Optimization problem

Raises **AttributeError** – If object doesn't follow the pySOT standard

pySOT.utils.**from_unit_box**(*x*, *data*)
Maps a set of points from the unit box to the original domain

Parameters

- **x** (*numpy.array*) – Points to be mapped from the unit box, of size npts x dim
- **data** (*Object*) – Optimization problem, needs to have attributes xlow and xup

Returns Points mapped to the original domain

Return type numpy.array

pySOT.utils.**progress_plot**(*controller*, *title=''*, *interactive=False*)
Makes a progress plot from a POAP controller

This method depends on matplotlib and will terminate if matplotlib.pyplot is unavailable.

Parameters

- **controller** (*Object*) – POAP controller object
- **title** (*string*) – Title of the plot
- **interactive** (*bool*) – True if the plot should be interactive

pySOT.utils.**round_vars**(*data*, *x*)
Round integer variables to closest integer that is still in the domain

Parameters

- **data** (*Object*) – Optimization problem object
- **x** (*numpy.array*) – Set of points, of size npts x dim

Returns The set of points with the integer variables rounded to the closest integer in the domain

Return type numpy.array

pySOT.utils.**to_unit_box**(*x*, *data*)
Maps a set of points to the unit box

Parameters

- **x** (*numpy.array*) – Points to be mapped to the unit box, of size npts x dim
- **data** (*Object*) – Optimization problem, needs to have attributes xlow and xup

Returns Points mapped to the unit box

Return type numpy.array

pySOT.utils.**unit_rescale**(*xx*)
Shift and rescale elements of a vector to the unit interval

Parameters **xx** (*numpy.array*) – Vector that should be rescaled to the unit interval

Returns Vector scaled to the unit interval

Return type numpy.array

CHAPTER 10

Changes

v.0.1.36, 2017-07-20

- The GUI is now built in PyQt5 instead of PySide

v.0.1.35, 2017-04-29

- Added support for termination based on elapsed time
- Added the Hartman6 test problem

v.0.1.34, 2017-03-28

- Added support for adding points with known (and unknown) function values to the experimental design

v.0.1.33, 2016-12-27

- Fixed a bug in MARS that resulted in using a lot of zero points for fitting
- Added a GP regression object based on scikit-learn 0.18.1
- Updated tests and documentation

v.0.1.32, 2016-12-07

- Switched to make py-earth, matlab_wrapper, and subprocess32 optional dependencies to resolve pip installation issues

v.0.1.31, 2016-11-23

- Added Python 3 support
- Removed Sphinx dependency
- Added six dependency to get py-earth to work for Python 3

v.0.1.30, 2016-11-18

- Moved all of the official pySOT documentation over to Sphinx
- Five pySOT tutorials were added to the documentation
- The documentation is now hosted on Read the Docs (https://pysot.readthedocs.io)
- Removed pyKriging in order to remove the matplotlib and inspyred dependencies. A new Kriging module will be added in the next version.
- Added the MARS installation to the setup.py since it can now be installed via scikit-learn
- Updated the Sphinx documentation to include all of the source files
- The License, Changes, Contributors, and README files are not in .rst
- Renamed sampling_methods.py to adaptive_sampling.py
- Moved the kernels and tails to separate Python files
- Added a Gitter for pySOT

v0.1.29, 2016-10-20

- Correcting an error in the pypi upload

v0.1.28, 2016-10-20

- Making the GUI work with the new RBF design

v0.1.27, 2016-10-18

- Removed dimensionality argument for the RBF to match the other surrogates

v0.1.26, 2016-10-14

- Signficant changes in the RBFInterpolant. Users need to update their code
- Added RBF regression surfaces
- Added version information in the module. pySOT.__version__ gives the version of the current pySOT installation

- The Gutmann strategy has been temporarily removed due to the RBF redesign, but will be added back soon
- Check out test_rbf.py to see how to use the new RBF

v0.1.25, 2016-09-14

- Fixed a bug in DYCORS when the subset has length 1

v0.1.24, 2016-08-04

- Changed to setup.py to use rst format for pypi

v0.1.23, 2016-07-28

- Updates to support the new MPIController in POAP
- pySOT now sends copies of key variables in case they are changed by the method

v0.1.23, 2016-07-28

- Updates to support the new MPIController in POAP
- pySOT now sends copies of key variables in case they are changed by the method

v0.1.22, 2016-06-27

- Added two tests for the MPI controller in POAP
- Removed the accidental matplotlib dependency
- Fixed some printouts in the tests

v0.1.21, 2016-06-23

- Added an option for supplying weights to the candidate point methods
- Cleaned up some of the tests by appending attributes to the workers
- Extended the MATLAB example to parallel
- Added a help function for doing a progress plot

v0.1.20, 2016-06-18

- Added some basic input checking (evaluations, dimensionality, etc)
- Added an example with a MATLAB engine in case the optimization problems is in MATLAB
- Fixed a bug in the polynomial regression
- Moved the merit function out of sampling_methods.py

v0.1.19, 2016-01-30

- Too much regularization was added to the RBF surface when the volume of the domain was large. This has been fixed.

v0.1.18, 2016-01-24

- Significant restructuring of the code base
- make_points now takes an argument that specifies the number of new points to be generated
- Added Box-Behnken and 2-factorial to the experimental designs
- Simplified the penalty method strategy by moving evals and derivs into a surrogate wrapper

v0.1.17, 2016-01-13

- Added the possibility to input the penalty for the penalty method in the GUI
- Added the possibility of making a performance plot using matplotlib that adds new points dynamically as evaluations are finished
- Switched from subprocess to subprocess32

v0.1.16, 2016-01-06

- Added a projection strategy

v0.1.15, 2015-09-23

- Added an example test_subprocess_files that shows how to use pySOT in case the objective function needs to read the input from a textfile

v0.1.14, 2015-09-22

- Updated the Tutorial to reflect the changes for the last few months
- Simplified the object creation from strings in the GUI by importing directly from the namespace.

v0.1.13, 2015-09-03

- Allowed to still import the rest of pySOT when PySide is not found. In this case, the GUI will be unavailable.

v0.1.12, 2015-07-23

- The capping can now take in a general transformation that is used to transform the function values. Default is median capping.
- The Genetic Algorithm now defaults to initialize the population using a symmetric latin hypercube design
- DYCORS uses the remaining evaluation budget to change the probabilities after a restart instead of using the total budget

v0.1.11, 2015-07-22

- Fixed a bug in the capped response surface
- pySOT now internally works on the unit hypercube
- The distance can be passed to the RBF after being computed when generating candidate points so it's not computed twice anymore
- Fixed some bugs in the candidate functions
- GA and Multi-Search gradient perturb the best solution in the case when the best solution is a previously evaluated point
- Added an additional test for the multi-search strategy

v0.1.10, 2015-07-14

- README.md not uploaded to pypi which caused pip install to fail

v0.1.9, 2015-07-13

- Fixed a bug in the merit function and several bugs in the DYCORS strategy
- Added a DDS candidate based strategy for searching on the surrogate

v0.1.8, 2015-07-01

- Multi Start Gradient method that uses the L-BFGS-B algorithm to search on the surroagate

v0.1.7, 2015-06-30

- Fixed some parameters (and bugs) to improve the DYCORS results. Using DYCORS together with the genetic algorithm is recommended.
- Added polynomial regression (not yet in the GUI)
- Changed so that candidate points are generated using truncated normal distribution to avoid projections onto the boundary
- Removed some accidental scikit dependencies in the ensemble surrogate

v0.1.6, 2015-06-28

- GUI inactivates all buttons but the stop button while running
- Bug fixes

v0.1.5, 2015-06-28

- GUI now has support for multiple search strategies and ensemble surrogates
- Reallocation bug in the ensemble surrogates fixed
- Genetic algorithm added to search on the surrogate

v0.1.4, 2015-06-26

- GUI now has improved error handling
- Strategies informs the user if they get constraints when not expecting constraints (and the other way) before the run starts

v0.1.3, 2015-06-26

- Experimental (but not documented) GUI added. You need PySide to use it.
- Changes in testproblems.py to allow external objective functions that implement ProcessWorkerThread
- Added GUI test examples in documentation (Ackley.py, Keane.py, SphereExt.py)

v0.1.2, 2015-06-24

- Changed to using the logging module for all the logging in order to conform to the changes in POAP 0.1.9
- The quiet and stream arguments in the strategies were removed and the tests updated accordingly
- Turned sleeping of in the subprocess test, to avoid platform dependency issues

v0.1.1, 2015-06-21

- surrogate_optimizer removed, so the user now has to create his own controller
- constraint_method.py is gone, and the constraint handling is handled in specific strategies instead
- There are now two strategies, SyncStrategyNoConstraints and SyncStrategyPenalty
- The search strategies now take a method for providing surrogate predictions rather than keeping a copy of the response surface
- It is now possible for the user to provide additional points to be added to the initial design, in case a 'good starting point' is known.
- Ensemble surrogates have been added to the toolbox
- The strategies takes an additional option 'quiet' so that all of the printing can be avoided if the user wants
- There is also an option 'stream' in case the printing should be redirected somewhere else, for example to a text file. Default is printing to stdout.
- Several examples added to pySOT.test

v0.1.0, 2015-06-03

- Initial release

CHAPTER 11

License

CHAPTER 12

Contributors

Developed and maintained by:

- David Bindel <bindel@cs.cornell.edu>
- David Eriksson <dme65@cornell.edu>
- Christine Shoemaker <cas12@cornell.edu>

with contributions by:

- Yi Shen <ys623@cornell.edu>

Python Module Index

Index

E

F

G

H

I

K

L

M

O

P

Q

R

S

T

U

W