# User Documentation

## 1 Quickstart

1.1 Dependencies ................................................. 3  
1.2 Installation ................................................. 3

## 2 Surrogate optimization ............................................. 5

## 3 Options ................................................................ 7

3.1 Strategy ......................................................... 7  
3.1.1 SRBFStrategy ............................................... 7  
3.1.2 DYCORSStrategy ......................................... 8  
3.1.3 EIStrategy ................................................... 9  
3.1.4 LCBStrategy ............................................... 9
  
3.2 Experimental design ........................................... 10  
3.2.1 LatinHypercube .......................................... 10  
3.2.2 SymmetricLatinHypercube ............................. 11  
3.2.3 TwoFactorial .............................................. 11
  
3.3 Surrogate model ................................................ 11  
3.3.1 RBFInterpolant ........................................... 12  
3.3.2 GPregressor ............................................... 12  
3.3.3 MARSInterpolant .......................................... 13  
3.3.4 PolyRegress .............................................. 13
  
3.4 Optimization problem ......................................... 14

## 4 POAP ................................................................. 15

4.1 Controller ...................................................... 15  
4.2 Strategy ......................................................... 16  
4.3 Workers ......................................................... 16  
4.4 Communication between POAP and pySOT ................. 17

## 5 Logging ................................................................ 19

## 6 Source code .......................................................... 21

6.1 pySOT.auxiliary_problems module .......................... 21  
6.2 pySOT.controller module ...................................... 25  
6.3 pySOT.experimental_design module ......................... 26  
6.4 pySOT.optimization_problems module ...................... 28  
6.5 pySOT.strategy module ....................................... 37
7 Changes

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.0</td>
<td>v.0.2.2, 2019-02-12</td>
</tr>
<tr>
<td>7.1</td>
<td>v.0.2.1, 2019-01-26</td>
</tr>
<tr>
<td>7.2</td>
<td>v.0.2.0, 2018-12-06</td>
</tr>
<tr>
<td>7.3</td>
<td>v.0.1.36, 2017-07-20</td>
</tr>
<tr>
<td>7.4</td>
<td>v.0.1.35, 2017-04-29</td>
</tr>
<tr>
<td>7.5</td>
<td>v.0.1.34, 2017-03-28</td>
</tr>
<tr>
<td>7.6</td>
<td>v.0.1.33, 2016-12-27</td>
</tr>
<tr>
<td>7.7</td>
<td>v.0.1.32, 2016-12-07</td>
</tr>
<tr>
<td>7.8</td>
<td>v.0.1.31, 2016-11-23</td>
</tr>
<tr>
<td>7.9</td>
<td>v.0.1.30, 2016-11-18</td>
</tr>
<tr>
<td>7.10</td>
<td>v.0.1.29, 2016-10-20</td>
</tr>
<tr>
<td>7.11</td>
<td>v.0.1.28, 2016-10-20</td>
</tr>
<tr>
<td>7.12</td>
<td>v.0.1.27, 2016-10-18</td>
</tr>
<tr>
<td>7.13</td>
<td>v.0.1.26, 2016-10-14</td>
</tr>
<tr>
<td>7.14</td>
<td>v.0.1.25, 2016-09-14</td>
</tr>
<tr>
<td>7.15</td>
<td>v.0.1.24, 2016-08-04</td>
</tr>
<tr>
<td>7.16</td>
<td>v.0.1.23, 2016-07-28</td>
</tr>
<tr>
<td>7.17</td>
<td>v.0.1.23, 2016-07-28</td>
</tr>
<tr>
<td>7.18</td>
<td>v.0.1.22, 2016-06-27</td>
</tr>
<tr>
<td>7.19</td>
<td>v.0.1.21, 2016-06-23</td>
</tr>
<tr>
<td>7.20</td>
<td>v.0.1.20, 2016-06-18</td>
</tr>
<tr>
<td>7.21</td>
<td>v.0.1.19, 2016-01-30</td>
</tr>
<tr>
<td>7.22</td>
<td>v.0.1.18, 2016-01-24</td>
</tr>
<tr>
<td>7.23</td>
<td>v.0.1.17, 2016-01-13</td>
</tr>
<tr>
<td>7.24</td>
<td>v.0.1.16, 2016-01-06</td>
</tr>
<tr>
<td>7.25</td>
<td>v.0.1.15, 2015-09-23</td>
</tr>
<tr>
<td>7.26</td>
<td>v.0.1.14, 2015-09-22</td>
</tr>
<tr>
<td>7.27</td>
<td>v.0.1.13, 2015-09-03</td>
</tr>
<tr>
<td>7.28</td>
<td>v.0.1.12, 2015-07-23</td>
</tr>
<tr>
<td>7.29</td>
<td>v.0.1.11, 2015-07-22</td>
</tr>
<tr>
<td>7.30</td>
<td>v.0.1.10, 2015-07-14</td>
</tr>
<tr>
<td>7.31</td>
<td>v.0.1.9, 2015-07-13</td>
</tr>
<tr>
<td>7.32</td>
<td>v.0.1.8, 2015-07-01</td>
</tr>
<tr>
<td>7.33</td>
<td>v.0.1.7, 2015-06-30</td>
</tr>
<tr>
<td>7.34</td>
<td>v.0.1.6, 2015-06-28</td>
</tr>
<tr>
<td>7.35</td>
<td>v.0.1.5, 2015-06-28</td>
</tr>
<tr>
<td>7.36</td>
<td>v.0.1.4, 2015-06-26</td>
</tr>
<tr>
<td>7.37</td>
<td>v.0.1.3, 2015-06-26</td>
</tr>
<tr>
<td>7.38</td>
<td>v.0.1.2, 2015-06-24</td>
</tr>
<tr>
<td>7.39</td>
<td>v.0.1.1, 2015-06-21</td>
</tr>
<tr>
<td>7.40</td>
<td>v.0.1.0, 2015-06-03</td>
</tr>
<tr>
<td>7.41</td>
<td>v.0.1.0, 2015-06-03</td>
</tr>
</tbody>
</table>

8 License

9 Contributors

Python Module Index
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. 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. 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]
1.1 Dependencies

Before starting you will need Python 3.4 or newer. 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. **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
   ```

2. **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
   ```

1.2 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:

   ```bash
   pip install pySOT
   ```

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

2.1. Clone the repository:

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

2.2. Navigate to the repository using:

   ```bash
   cd pySOT
   ```

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

   ```bash
   python setup.py install
   ```

Several examples are available in ./pySOT/examples and ./pySOT/notebooks
Surrogate optimization algorithms generally consist of four components:

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

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.

The general framework for a Surrogate Optimization algorithm is illustrated in the algorithm below:

**Inputs**: Optimization problem, Experimental design, Optimization strategy, Surrogate model, Stopping criterion

```
1. Generate an initial experimental design
2. Evaluate the points in the experimental design
3. Build a Surrogate model from the data
4. Repeat until stopping criterion met
   5. Use the strategy to generate new point(s) to evaluate
      6. Evaluate the point(s) generated using all computational resources
      7. 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

3.1 Strategy

We provide implementations of Stochastic RBF (SRBF), DYCORS, Expected Improvement (EI), lower confidence bound (LCB) and random search (RS). EI can only be used in combination with GPRegressor since uncertainty predictions are necessary. All strategies support running in serial, batch synchronous parallel, and asynchronous parallel.

New optimization strategies can be implemented by inheriting from SurrogateBaseStrategy and implementing the abstract generate_evals method that proposes num_pts new sample points:

- Required methods
  - generate_evals(num_pts): Proposes num_pts new samples.

The following strategies are currently supported:

3.1.1 SRBFStrategy

This is an implementation of the SRBF strategy by Regis and Shoemaker:

Rommel G Regis and Christine A Shoemaker.
A stochastic radial basis function method for the global optimization of expensive functions.

Rommel G Regis and Christine A Shoemaker. Parallel stochastic global optimization using radial basis functions.

The main idea is to pick the new evaluations from a set of candidate points where each candidate point is generated as an N(0, sigma^2) distributed perturbation from the current best solution. The value of sigma is modified based on progress and follows the same logic as in many trust region methods; we increase sigma if we make a lot of progress (the surrogate is accurate) and decrease sigma when we aren’t able to make progress (the surrogate model is inaccurate). More details about how sigma is updated is given in the original papers.
After generating the candidate points we predict their objective function value and compute the minimum distance to previously evaluated point. Let the candidate points be denoted by $C$ and let the function value predictions be $s(x_i)$ and the distance values be $d(x_i)$, both rescaled through a linear transformation to the interval $[0,1]$. This is done to put the values on the same scale. The next point selected for evaluation is the candidate point $x$ that minimizes the weighted-distance merit function:

$$merit(x) := ws(x) + (1 - w)(1 - d(x))$$

where $0 \leq w \leq 1$. That is, we want a small function value prediction and a large minimum distance from previously evaluated points. The weight $w$ is commonly cycled between a few values to achieve both exploitation and exploration. When $w$ is close to zero we do pure exploration while $w$ close to 1 corresponds to exploitation.

- **Parameters:**
  - max_evals: Evaluation budget (int)
  - opt_prob: Optimization problem object, must implement OptimizationProblem
  - exp_design: Experimental design object, must implement ExperimentalDesign
  - surrogate: Surrogate object, must implement Surrogate
  - asynchronous: Whether or not to use asynchrony (True / False).
  - batch_size: Size of the batch. This value is ignored if asynchronous is True. Use 1 for serial or run with asynchronous set to True.
  - extra_points: n Extra points to add to the experimental design (numpy.array of size n x dim)
  - extra_vals: Values for extra_points. Set elements to np.nan if unknown (numpy.array of size n x 1)
  - reset_surrogate: Specify whether or not we are resetting the surrogate model i.e., removing current points (True / False)
  - weights: Weights for merit function (list or numpy.array). Default is [0.3, 0.5, 0.8, 0.95]
  - num_cand: Number of candidate points (int). Default = 100*dim

### 3.1.2 DYCORSStrategy

This is an implementation of the DYCORS strategy by Regis and Shoemaker:

Rommel G Regis and Christine A Shoemaker.
Combining radial basis function surrogates and dynamic coordinate search in high-dimensional expensive black-box optimization.

This is an extension of the SRBF strategy that changes how the candidate points are generated. The main idea is that many objective functions depend only on a few directions so it may be advantageous to perturb only a few directions. In particular, we use a perturbation probability to perturb a given coordinate and decrease this probability after each function evaluation so fewer coordinates are perturbed later in the optimization.

The parameters are the same as in the SRBF strategy.
3.1.3 EIStrategy

This is an implementation of Expected Improvement (EI), arguably the most popular acquisition function in Bayesian optimization. Under a Gaussian process (GP) prior, the expected value of the improvement:

\[ I(x) := \max(f_{\text{best}} - f(x), 0) \]
\[ \text{EI}[x] := E[I(x)] \]

can be computed analytically, where \( f_{\text{best}} \) is the best observed function value. EI is one-step optimal in the sense that selecting the maximizer of EI is the optimal action if we have exactly one function value remaining and must return a solution with a known function value.

When using parallelism, we constrain each new evaluation to be a distance \( \text{dtol} \) away from previous and pending evaluations to avoid that the same point is being evaluated multiple times. We use a default value of \( \text{dtol} = 1\times 10^{-3} \times \|\text{ub} - \text{lb}\| \), but note that this value has not been tuned carefully and may be far from optimal.

The optimization strategy terminates when the evaluation budget has been exceeded or when the EI of the next point falls below some threshold, where the default threshold is \( 1\times 10^{-6} \times (\max(fX) - \min(fX)) \).

- **Parameters:**
  - max_evals: Evaluation budget (int)
  - opt_prob: Optimization problem object, must implement OptimizationProblem
  - exp_design: Experimental design object, must implement ExperimentalDesign
  - surrogate: Surrogate object, must implement Surrogate
  - asynchronous: Whether or not to use asynchrony (True / False).
  - batch_size: Size of the batch. This value is ignored if asynchronous is True. Use 1 for serial or run with asynchronous set to True.
  - extra_points: n Extra points to add to the experimental design (numpy.array of size n x dim)
  - extra_vals: Values for extra_points. Set elements to np.nan if unknown (numpy.array of size n x 1)
  - reset_surrogate: Specify whether or not we are resetting the surrogate model i.e., removing current points (True / False)
  - ei_tol: Terminate if the largest EI falls below this threshold (float). Default: \( 1\times 10^{-6} \times (\max(fX) - \min(fX)) \)
  - dtol: Minimum distance between new and pending/finished evaluations (float). Default: \( 1\times 10^{-3} \times \|\text{ub} - \text{lb}\| \)

3.1.4 LCBStrategy

This is an implementation of Lower Confidence Bound (LCB), a popular acquisition function in Bayesian optimization. The main idea is to minimize:

\[ \text{LCB}(x) := E[x] - \kappa \sqrt{V[x]} \]

where \( E[x] \) is the predicted function value, \( V[x] \) is the predicted variance, and kappa is a constant that balances exploration and exploitation. We use a default value of kappa = 2.

When using parallelism, we constrain each new evaluation to be a distance \( \text{dtol} \) away from previous and pending evaluations to avoid that the same point is being evaluated multiple times. We use a default value of \( \text{dtol} = 1\times 10^{-3} \times \|\text{ub} - \text{lb}\| \), but note that this value has not been tuned carefully and may be far from optimal.

The optimization strategy terminates when the evaluation budget has been exceeded or when the LCB of the next point falls below some threshold, where the default threshold is \( 1\times 10^{-6} \times (\max(fX) - \min(fX)) \).
• Parameters:
  – max_evals: Evaluation budget (int)
  – opt_prob: Optimization problem object, must implement OptimizationProblem
  – exp_design: Experimental design object, must implement ExperimentalDesign
  – surrogate: Surrogate object, must implement Surrogate
  – asynchronous: Whether or not to use asynchrony (True / False).
  – batch_size: Size of the batch. This value is ignored if asynchronous is True. Use 1 for serial or run with asynchronous set to True.
  – extra_points: n Extra points to add to the experimental design (numpy.array of size n x dim)
  – extra_vals: Values for extra_points. Set elements to np.nan if unknown (numpy.array of size n x 1)
  – reset_surrogate: Specify whether or not we are resetting the surrogate model i.e., removing current points (True / False)
  – kappa: Constant in the LCB merit function (float). Default: 2.0
  – lcb_tol: Terminate if min(fX) - min(LCB(x)) < lcb_tol (float). Default: 1e-6 * (max(fX) - min(fX))
  – dtol: Minimum distance between new and pending/finished evaluations (float). Default: 1e-3 * norm(ub - lb)

3.2 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
  – num_pts: Number of points in the design

• Required methods
  – generate_points(lb, ub, int_var): Returns an experimental design of size num_pts x dim where num_pts is the number of points in the initial design, which was specified when the object was created. You can supply lb, ub, and int_var to have the design mapped before it’s scored instead of having the rounding take place in the strategy.

The following experimental designs are supported:

3.2.1 LatinHypercube

A Latin hypercube design

• Parameters:
  – dim: Number of dimensions (int).
  – num_pts: Number of desired sampling points (int).
  – iterations: Number of designs to generate and choose the best from (int)

Example:
```python
from pySOT import LatinHypercube
exp_des = LatinHypercube(dim=3, num_pts=10)
```

creates a Latin hypercube design with 10 points in 3 dimensions

### 3.2.2 SymmetricLatinHypercube

A symmetric Latin hypercube design

- **Parameters:**
  - dim: Number of dimensions (int).
  - num_pts: Number of desired sampling points (int). Use 2*dim + 1 to make sure the design has full rank.
  - iterations: Number of designs to generate and choose the best from (int)

Example:

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

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

### 3.2.3 TwoFactorial

The corners of the unit hypercube

- **Parameters:**
  - dim: Number of dimensions (int).

Example:

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

creates a two factorial design with 8 points in 3 dimensions

### 3.3 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:**
  - dim: Number of dimensions
  - num_pts: Number of points in the surrogate model
  - X: Data points, of size num_pts x dim, currently incorporated in the model
  - fX: Function values at the data points
  - updated: True if all information is incorporated in the model, else a new fit will be triggered

- **Required methods**
- reset(): Resets the surrogate model
- add_points(x, fx): Adds point(s) x with value(s) fx to the surrogate model. This SHOULD NOT trigger a new fit of the model.
- predict(x): Evaluates the surrogate model at points x
- predict_deriv(x): Evaluates the derivative of surrogate model at points x

**Optional methods**
- predict_std(x): Evaluates the uncertainty of the surrogate model at points x

The following surrogate models are supported:

### 3.3.1 RBFInterpolant

A radial basis function (RBF) takes the form:

\[ s(x) = \sum_j c_j \phi(||x - x_j||) + \sum_j \lambda_j p_j(x) \]

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 \( \eta \) allows us to avoid problems with potential poor conditioning of the system. Consider using the SurrogateUnitBox wrapper or manually scaling the domain to the unit hypercube to avoid issues with the domain scaling.

We add k new points to the RBFInterpolant in \( O(kn^2) \) flops by updating the LU factorization of the old RBF system. This is better than computing the RBF coefficients from scratch, which costs \( O(n^3) \) flops.

**Parameters:**
- dim: Number of dimensions (int)
- kernel: RBF kernel object, must implement Kernel. Default: CubicKernel()
- tail: RBF polynomial tail object, must implement Tail. Default: LinearTail(dim)
- eta: Regularization parameter. Use something small like 1e-6 if the domain is \([0, 1]^d\)

Example:

```python
from pySOT.surrogate import RBFInterpolant, CubicKernel, LinearTail
fhat = RBFInterpolant(dim=dim, kernel=CubicKernel(), tail=LinearTail(dim=dim))
```

creates a cubic RBF with a linear tail in dim dimensions.

### 3.3.2 GPRegressor

Generate a Gaussian process regression object. This is just a wrapper around the GPRegressor in scikit-learn.

**Parameters:**
- dim: Number of dimensions (int)
- gp: GPRegressor model in scikit-learn. Uses the SE/RBF/Gaussian kernel as a default if None is passed.
n_restarts_optimizer: Number of restarts in hyperparameter fitting (int)

Example:
```
from pySOT.surrogate import GPRegressor
surrogate = GPRegressor(dim=dim)
```

creates a GPRegressor object in dim dimensions.

### 3.3.3 MARSInterpolant

Generate a Multivariate Adaptive Regression Splines (MARS) model.

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

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 \( \max(0, x - \text{const}) \) or \( \max(0, \text{const} - x) \). 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 can model interaction between two or more variables.

- **Parameters:**
  - dim: Number of dimensions (int)

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

Example:
```
from pySOT.surrogate import MARSInterpolant
surrogate = MARSInterpolant(dim=dim)
```

creates a MARS interpolant in dim dimensions.

### 3.3.4 PolyRegressor

Multivariate polynomial regression with cross-terms. This is just a wrapper around PolynomialFeatures in scikit-learn.

- **Parameters:**
  - dim: Number of dimensions (int)
  - degree: Polynomial degree (int)

Example:
```
from pySOT.surrogate import PolyRegressor
surrogate = PolyRegressor(dim=dim, degree=2)
```

creates a polynomial regressor of degree 2.
3.4 Optimization problem

The optimization problem is its own object and must have certain attributes and methods in order to work with the framework. The following attributes and methods must always be specified in the optimization problem class:

- **Attributes**
  - `lb`: Lower bounds for the variables.
  - `ub`: Upper bounds for the variables.
  - `dim`: Number of dimensions
  - `int_var`: Specifies the integer variables. If no variables have discrete, set to `[]`
  - `cont_var`: Specifies the continuous variables. If no variables are continuous, set to `[]`

- **Required methods**
  - `eval`: Takes one input in the form of a 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.
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:

```python
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)
```

### 4.1 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.

4.2 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.

4.3 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 con-
troller. 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.
4.4 Communication between POAP and pySOT
CHAPTER 5

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.
6.1 pySOT.auxiliary_problems module

Module auxiliary_problems

Author David Eriksson <dme65@cornell.edu>.

pySOT.auxiliary_problems.candidate_dycors(num_pts, opt_prob, surrogate, X, fX, weights, prob_perturb, Xpend=None, sampling_radius=0.2, subset=None, dtol=0.001, num_cand=None)

Select new evaluations using DYCORS.

Parameters

- **num_pts**(int) – Number of points to generate
- **opt_prob**(object) – Optimization problem
- **surrogate**(object) – Surrogate model object
- **X**(numpy.array) – Previously evaluated points, of size n x dim
- **fX**(numpy.array) – Values at previously evaluated points, of size n x 1
- **weights**(list or numpy.array) – num_pts weights in [0, 1] for merit function
- **prob_perturb**(list or numpy.array) – Probability to perturb a given coordinate
- **Xpend**(numpy.array) – Pending evaluations
- **sampling_radius**(float) – Perturbation radius
- **subset**(list or numpy.array) – Coordinates that should be perturbed, use None for all
- **dtol**(float) – Minimum distance between evaluated and pending points
- **num_cand**(int) – Number of candidate points
pySOT Documentation

Returns The num_pts new points to evaluate

Return type numpy.array of size num_pts x dim

pySOT.auxiliary_problems.candidate_srbf(num_pts, opt_prob, surrogate, X, fX, weights, Xpend=None, sampling_radius=0.2, subset=None, dtol=0.001, num_cand=None)

Select new evaluations using Stochastic RBF (SRBF).

Parameters

- num_pts (int) – Number of points to generate
- opt_prob (object) – Optimization problem
- surrogate (object) – Surrogate model object
- X (numpy.array) – Previously evaluated points, of size n x dim
- fX (numpy.array) – Values at previously evaluated points, of size n x 1
- weights (list or numpy.array) – num_pts weights in [0, 1] for merit function
- Xpend (numpy.array) – Pending evaluation, of size k x dim
- sampling_radius (float) – Perturbation radius
- subset (list or numpy.array) – Coordinates that should be perturbed, use None for all
- dtol (float) – Minimum distance between evaluated and pending points
- num_cand (int) – Number of candidate points

Returns The num_pts new points to evaluate

Return type numpy.array of size num_pts x dim

Select new evaluations from uniform candidate points.

Parameters

- num_pts (int) – Number of points to generate
- opt_prob (object) – Optimization problem
- surrogate (object) – Surrogate model object
- X (numpy.array) – Previously evaluated points, of size n x dim
- fX (numpy.array) – Values at previously evaluated points, of size n x 1
- weights (list or numpy.array) – num_pts weights in [0, 1] for merit function
- Xpend (numpy.array) – Pending evaluations
- subset (list or numpy.array) – Coordinates that should be perturbed, use None for all
- dtol (float) – Minimum distance between evaluated and pending points
- num_cand (int) – Number of candidate points

Returns The num_pts new points to evaluate

Return type numpy.array of size num_pts x dim
pySOT.auxiliary_problems.ei_merit \((X, \text{surrogate}, fX, XX=None, dtol=0)\)
Compute the expected improvement merit function.

Parameters
- \(X\) (numpy.array) – Points where to compute EI, of size \(n \times \text{dim}\)
- \text{surrogate\} (object) – Surrogate model object, must implement predict_std
- \(fX\) (numpy.array) – Values at previously evaluated points, of size \(m \times 1\)
- \(XX\) (numpy.array) – Previously evaluated points, of size \(m \times 1\)
- \(dtol\) (float) – Minimum distance between evaluated and pending points

Returns Evaluate the expected improvement for points \(X\)
Return type numpy.array of length \(X\).shape[0]

pySOT.auxiliary_problems.expected_improvement_ga \((\text{num_pts, opt_prob, surrogate, } X, fX, Xpend=None, dtol=0.001, ei_tol=1e-06)\)
Maximize EI using a genetic algorithm.

Parameters
- \text{num_pts} (int) – Number of points to generate
- \text{opt_prob} (object) – Optimization problem
- \text{surrogate} (object) – Surrogate model object
- \(X\) (numpy.array) – Previously evaluated points, of size \(n \times \text{dim}\)
- \(fX\) (numpy.array) – Values at previously evaluated points, of size \(n \times 1\)
- \(Xpend\) (numpy.array) – Pending evaluations
- \(dtol\) (float) – Minimum distance between evaluated and pending points
- \(ei\_tol\) (float) – Return None if we don’t find an EI of at least this value

Returns \text{num_pts} new points to evaluate
Return type numpy.array of size \text{num_pts} \times \text{dim}

pySOT.auxiliary_problems.expected_improvement_uniform \((\text{num_pts, opt_prob, surrogate, } X, fX, Xpend=None, dtol=0.001, ei\_tol=1e-06, \text{num_cand=None})\)
Maximize EI from a uniform set of points.

Parameters
- \text{num_pts} (int) – Number of points to generate
- \text{opt_prob} (object) – Optimization problem
- \text{surrogate} (object) – Surrogate model object
- \(X\) (numpy.array) – Previously evaluated points, of size \(n \times \text{dim}\)
- \(fX\) (numpy.array) – Values at previously evaluated points, of size \(n \times 1\)
- \(Xpend\) (numpy.array) – Pending evaluations
- \(dtol\) (float) – Minimum distance between evaluated and pending points
- \(ei\_tol\) (float) – Return None if we can’t reach this threshold

6.1. pySOT.auxiliary_problems module
• **num_cand** (*int*) – Number of candidate points

**Returns** num_pts new points to evaluate

**Return type** numpy.array of size num_pts x dim

**pySOT.auxiliary_problems.lcb_merit** (*X, surrogate, fX, XX=None, dtol=0.0, kappa=2.0*)
Compute the lcb merit function.

**Parameters**

- **X** (*numpy.array*) – Points where to compute LCB, of size n x dim
- **surrogate** (*object*) – Surrogate model object, must implement predict_std
- **fX** (*numpy.array*) – Values at previously evaluated points, of size m x 1
- **XX** (*numpy.array*) – Previously evaluated points, of size m x 1
- **dtol** (*float*) – Minimum distance between evaluated and pending points
- **kappa** (*float*) – Constant in front of standard deviation Default: 2.0

**Returns** Evaluate the lower confidence bound for points X

**Return type** numpy.array of length X.shape[0]

**pySOT.auxiliary_problems.lower_confidence_bound_ga** (*num_pts, opt_prob, surrogate, X, fX, Xpend=None, kappa=2.0, dtol=0.001, lcb_target=None*)
Minimize the LCB using a genetic algorithm.

**Parameters**

- **num_pts** (*int*) – Number of points to generate
- **opt_prob** (*object*) – Optimization problem
- **surrogate** (*object*) – Surrogate model object
- **X** (*numpy.array*) – Previously evaluated points, of size n x dim
- **fX** (*numpy.array*) – Values at previously evaluated points, of size n x 1
- **Xpend** (*numpy.array*) – Pending evaluations
- **dtol** (*float*) – Minimum distance between evaluated and pending points
- **lcb_target** (*float*) – Return None if we don’t find an LCB value <= lcb_target

**Returns** num_pts new points to evaluate

**Return type** numpy.array of size num_pts x dim

**pySOT.auxiliary_problems.weighted_distance_merit** (*num_pts, surrogate, X, fX, cand, weights, Xpend=None, dtol=0.001*)
Compute the weighted distance merit function.

**Parameters**

- **num_pts** (*int*) – Number of points to generate
- **surrogate** (*object*) – Surrogate model object
- **X** (*numpy.array*) – Previously evaluated points, of size n x dim
- **fX** (*numpy.array*) – Values at previously evaluated points, of size n x 1
- **cand** (*numpy.array*) – Candidate points to select from, of size m x dim
• **weights** *(list or numpy.array)* – num_pts weights in [0, 1] for merit function
• **Xpend** *(numpy.array)* – Pending evaluation, of size k x dim
• **dtol** *(float)* – Minimum distance between evaluated and pending points

**Returns**  The num_pts new points chosen from the candidate points

**Return type**  numpy.array of size num_pts x dim

## 6.2 pySOT.controller module

**Module**  controller

**Author**  David Eriksson <dme65@cornell.edu>,

**class pySOT.controller.CheckpointController** *(controller, fname='checkpoint.pysot')*

Checkpoint controller

Controller that uses dill to take snapshots of the strategy each time an evaluation is completed, killed, or the run is terminated. We assume that the strategy can be pickled, or this won’t work. We currently do not respect potential termination callbacks and failed evaluation callbacks. The strategy needs to implement a resume method that is called when a run is resumed. The strategy object can assume that all pending evaluations have been killed and that their respective callbacks won’t be executed

**Parameters**

• **controller** *(Controller)* – POAP controller
• **fname** *(string)* – Filename for checkpoint file (file cannot exist for new run)

**Variables**

• **controller** – POAP controller
• **fname** – Filename for snapshot

**on_cancel** *(record)*

“Handle record cancelled.

**Parameters**  **record** *(EvalRecord)* – Evaluation record

**on_complete** *(record)*

Handle feval completion.

**Parameters**  **record** *(EvalRecord)* – Evaluation record

**on_kill** *(record)*

“Handle record killed.

**Parameters**  **record** *(EvalRecord)* – Evaluation record

**on_new_feval** *(record)*

Handle new function evaluation request.

**Parameters**  **record** *(EvalRecord)* – Evaluation record

**on_terminate** *

“Handle termination.

**Parameters**  **record** *(EvalRecord)* – Evaluation record

**on_update** *(record)*

Handle feval update.

**Parameters**  **record** *(EvalRecord)* – Evaluation record
pySOT Documentation

```

```resume()
Resume an optimization run.

**Returns**
The record corresponding to the best solution

**Return type**
EvalRecord

```run()
Start the optimization run.

Make sure we do not overwrite any existing checkpointing files

**Returns**
The record corresponding to the best solution

**Return type**
EvalRecord

### 6.3 pySOT.experimental_design module

**Module** experimental_design

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

class pySOT.experimental_design.ExperimentalDesign
Base class for experimental designs.

**Variables**
- `dim` – Number of dimensions
- `num_pts` – Number of points in the experimental design

generate_points(lb=None, ub=None, int_var=None)

```class pySOT.experimental_design.LatinHypercube(dim, num_pts, criterion=None, iterations=1000)
Latin Hypercube experimental design.

**Parameters**
- `dim` (*int*) – Number of dimensions
- `num_pts` (*int*) – Number of desired sampling points
- `criterion` (*string*) – Previously passed to pyDOE, now deprecated
- `iterations` (*int*) – Number of designs to choose from

**Variables**
- `dim` – Number of dimensions
- `num_pts` – Number of points in the experimental design
- `iterations` – Number of points in the experimental design

generate_points(lb=None, ub=None, int_var=None)
Generate a new experimental design.

You can specify lb, ub, int_var to have the design mapped to a specific domain. These inputs are ignored if one of lb or ub is None. The design is generated in [0, 1]^d in this case.

**Parameters**
- `lb` (*numpy.array*) – Lower bounds
- `ub` (*numpy.array*) – Upper bounds
• `int_var(numpy.array)` – Indices of integer variables. If None, [], or np.array([]) we assume all variables are continuous.

Returns  Experimental design of size num_pts x dim

Return type  numpy.ndarray

class pySOT.experimental_design.SymmetricLatinHypercube(dim, num_pts, iterations=1000)
Symmetric Latin hypercube experimental design.

Parameters

• `dim(int)` – Number of dimensions
• `num_pts(int)` – Number of desired sampling points
• `iterations(int)` – Number of designs to generate and pick the best from

Variables

• `dim` – Number of dimensions
• `num_pts` – Number of points in the experimental design
• `iterations` – Number of points in the experimental design

generate_points(lb=None, ub=None, int_var=None)
Generate a new experimental design.

You can specify lb, ub, int_var to have the design mapped to a specific domain. These inputs are ignored if one of lb or ub is None. The design is generated in [0, 1]^d in this case.

Parameters

• `lb(numpy.array)` – Lower bounds
• `ub(numpy.array)` – Upper bounds
• `int_var(numpy.array)` – Indices of integer variables. If None, [], or np.array([]) we assume all variables are continuous.

Returns  Experimental design of size num_pts x dim

Return type  numpy.ndarray

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

Variables

• `dim` – Number of dimensions
• `num_pts` – Number of points in the experimental design

Raises  ValueError – If dim >= 15

generate_points(lb=None, ub=None, int_var=None)
Generate a two factorial design in the unit hypercube.

You can specify lb, ub, int_var to have the design mapped to a specific domain. These inputs are ignored if one of lb or ub is None. The design is generated in [0, 1]^d in this case.

Parameters
• **lb** (*numpy.array*) – Lower bounds

• **ub** (*numpy.array*) – Upper bounds

• **int_var** (*numpy.array*) – Indices of integer variables. If None, [], or np.array([]) we assume all variables are continuous.

**Returns** Two factorial design in unit hypercube of size num_pts x dim

**Return type** *numpy.array*

### 6.4 pySOT.optimization_problems module

**Module** optimization_problems

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

**class** `pySOT.optimization_problems.Ackley(dim=10)`

Ackley function

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

subject to

\[-15 \leq x_i \leq 20\]

Global optimum: \(f(0, 0, \ldots, 0) = 0\)

**Variables**

• **dim** – Number of dimensions

• **lb** – Lower variable bounds

• **ub** – Upper variable bounds

• **int_var** – Integer variables

• **cont_var** – Continuous variables

• **min** – Global minimum value

• **minimum** – Global minimizer

• **info** – String with problem info

**eval**(*x*)

Evaluate the Ackley function at \(x\)

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

**Returns** Value at \(x\)

**Return type** *float*

**class** `pySOT.optimization_problems.Branin`  

Branin function

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

Global optimum: \(f(-\pi, 12.275) = 0.397887\)

**Variables**
• **dim** – Number of dimensions
• **lb** – Lower variable bounds
• **ub** – Upper variable bounds
• **int_var** – Integer variables
• **cont_var** – Continuous variables
• **min** – Global minimum value
• **minimum** – Global minimizer
• **info** – String with problem info

```python
eval(x)
```
Evaluate the Branin function at `x`.

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

**Returns**
- Value at `x`

**Return type**
- float

```python
class pySOT.optimization_problems.Exponential(dim=10)
```
Exponential function

\[
f(x_1, \ldots, x_n) = \sum_{j=1}^{n} e^{x_j} - \sum_{j=1}^{n} e^{-5.12j}
\]

subject to

\[-5.12 \leq x_i \leq 5.12\]

Global optimum: \(f(0, 0, \ldots, 0) = 0\)

**Variables**
- **dim** – Number of dimensions
- **lb** – Lower variable bounds
- **ub** – Upper variable bounds
- **int_var** – Integer variables
- **cont_var** – Continuous variables
- **min** – Global minimum value
- **minimum** – Global minimizer
- **info** – String with problem info

```python
eval(x)
```
Evaluate the Exponential function at `x`.

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

**Returns**
- Value at `x`

**Return type**
- float

```python
class pySOT.optimization_problems.GoldsteinPrice
```
6.4. `pySOT.optimization_problems` module
**eval**<code>(x)</code>
Evaluate the Goldstein Price function at <code>x</code>

**Parameters**
- <code>x (numpy.array)</code> – Data point

**Returns**
Value at <code>x</code>

**Return type**
float

```python
class pySOT.optimization_problems.Griewank(dim=10)

Griewank function

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

subject to

\[-512 \leq x_i \leq 512\]

Global optimum: \(f(0, 0, \ldots, 0) = 0\)

**Variables**
- **dim** – Number of dimensions
- **lb** – Lower variable bounds
- **ub** – Upper variable bounds
- **int_var** – Integer variables
- **cont_var** – Continuous variables
- **min** – Global minimum value
- **minimum** – Global minimizer
- **info** – String with problem info

**eval**<code>(x)</code>
Evaluate the Griewank function at <code>x</code>.

**Parameters**
- <code>x (numpy.array)</code> – Data point

**Returns**
Value at <code>x</code>

**Return type**
float

```python
class pySOT.optimization_problems.Hartman3

Hartman 3 function

Details: [http://www.sfu.ca/~ssurjano/hart3.html](http://www.sfu.ca/~ssurjano/hart3.html)

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

**Variables**
- **dim** – Number of dimensions
- **lb** – Lower variable bounds
- **ub** – Upper variable bounds
- **int_var** – Integer variables
- **cont_var** – Continuous variables
- **min** – Global minimum value
• **minimum** – Global minimizer
• **info** – String with problem info

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

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

**Returns**
Value at *x*

**Return type**
float

class pySOT.optimization_problems.Hartman6
Hartman 6 function

Details: [http://www.sfu.ca/~ssurjano/hart6.html](http://www.sfu.ca/~ssurjano/hart6.html)

Global optimum: $f(0.201, 0.150, 0.476, 0.275, 0.311, 0.657) = -3.322$

**Variables**
- **dim** – Number of dimensions
- **lb** – Lower variable bounds
- **ub** – Upper variable bounds
- **int_var** – Integer variables
- **cont_var** – Continuous variables
- **min** – Global minimum value
- **minimum** – Global minimizer
- **info** – String with problem info

**eval**(*x*)
Evaluate the Hartman 6 function at *x*

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

**Returns**
Value at *x*

**Return type**
float

class pySOT.optimization_problems.Himmelblau(*dim=10*)
Himmelblau function

$$f(x_1, \ldots, x_n) = 10n - \frac{1}{2n} \sum_{i=1}^{n}(x_i^4 - 16x_i^2 + 5x_i)$$

Global optimum: $f(-2.903, \ldots, -2.903) = -39.166$

**Variables**
- **dim** – Number of dimensions
- **lb** – Lower variable bounds
- **ub** – Upper variable bounds
- **int_var** – Integer variables
- **cont_var** – Continuous variables
- **min** – Global minimum value
• **minimum** – Global minimizer
• **info** – String with problem info

`eval(x)`
Evaluate the Himmelblau function at x.

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

**Returns**
- Value at x

**Return type**
- float

```python
class pySOT.optimization_problems.Levy(dim=10)
Levy function
Details: https://www.sfu.ca/~ssurjano/levy.html
Global optimum: \( f(1, 1, \ldots, 1) = 0 \)

**Variables**
- **dim** – Number of dimensions
- **lb** – Lower variable bounds
- **ub** – Upper variable bounds
- **int_var** – Integer variables
- **cont_var** – Continuous variables
- **min** – Global minimum value
- **minimum** – Global minimizer
- **info** – String with problem info

`eval(x)`
Evaluate the Levy function at x.

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

**Returns**
- Value at x

**Return type**
- float

```python
class pySOT.optimization_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

\[0 \leq x_i \leq \pi\]

**Variables**
- **dim** – Number of dimensions
- **lb** – Lower variable bounds
- **ub** – Upper variable bounds
- **int_var** – Integer variables
- **cont_var** – Continuous variables
• **min** – Global minimum value
• **minimum** – Global minimizer
• **info** – String with problem info

def eval(x)
    Evaluate the Michalewicz function at x.
    Parameters:
    x (numpy.array) – Data point
    Returns:
    Value at x
    Return type: float

class pySOT.optimization_problems.OptimizationProblem
    Base class for optimization problems.

def eval(record)

class pySOT.optimization_problems.Perm(dim=10)
    Perm function
    Global optimum: \(f(1, 1/2, 1/3, \ldots, 1/n) = 0\)
    Variables
    • **dim** – Number of dimensions
    • **lb** – Lower variable bounds
    • **ub** – Upper variable bounds
    • **int_var** – Integer variables
    • **cont_var** – Continuous variables
    • **min** – Global minimum value
    • **minimum** – Global minimizer
    • **info** – String with problem info

def eval(x)
    Evaluate the Perm function at x.
    Parameters:
    x (numpy.array) – Data point
    Returns:
    Value at x
    Return type: float

class pySOT.optimization_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: \(f(0, 0, \ldots, 0) = 0\)
    Variables
    • **dim** – Number of dimensions
class pySOT.optimization_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: \( f(1, 1, \ldots, 1) = 0 \)

Variables
- \( \text{dim} \) – Number of dimensions
- \( \text{lb} \) – Lower variable bounds
- \( \text{ub} \) – Upper variable bounds
- \( \text{int\_var} \) – Integer variables
- \( \text{cont\_var} \) – Continuous variables
- \( \text{min} \) – Global minimum value
- \( \text{minimum} \) – Global minimizer
- \( \text{info} \) – String with problem info

eval(x)
Evaluate the Rosenbrock function at \( x \)

Parameters \( x (\text{numpy.array}) \) – Data point

Returns Value at \( x \)

Return type float

class pySOT.optimization_problems.Schwefel(dim=10)
Schwefel function

\[ f(x_1, \ldots, x_n) = \sum_{j=1}^{n} \left( -x_j \sin(\sqrt{|x_j|}) \right) + 418.982987n \]
subject to

\[-512 \leq x_i \leq 512\]

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

**Variables**

- \textbf{dim} – Number of dimensions
- \textbf{lb} – Lower variable bounds
- \textbf{ub} – Upper variable bounds
- \textbf{int\_var} – Integer variables
- \textbf{cont\_var} – Continuous variables
- \textbf{min} – Global minimum value
- \textbf{minimum} – Global minimizer
- \textbf{info} – String with problem info

\textbf{eval}(x)

Evaluate the Schwefel function at \(x\).

\textbf{Parameters}
\begin{itemize}
  \item \texttt{x (numpy.array)} – Data point
\end{itemize}

\textbf{Returns} Value at \(x\)

\textbf{Return type} float

\textbf{class} pySOT.optimization_problems.SixHumpCamel

Six-hump camel function

Details: https://www.sfu.ca/~ssurjano/camel6.html

Global optimum: \(f(0.0898, -0.7126) = -1.0316\)

**Variables**

- \textbf{dim} – Number of dimensions
- \textbf{lb} – Lower variable bounds
- \textbf{ub} – Upper variable bounds
- \textbf{int\_var} – Integer variables
- \textbf{cont\_var} – Continuous variables
- \textbf{min} – Global minimum value
- \textbf{minimum} – Global minimizer
- \textbf{info} – String with problem info

\textbf{eval}(x)

Evaluate the Six Hump Camel function at \(x\)

\textbf{Parameters}
\begin{itemize}
  \item \texttt{x (numpy.array)} – Data point
\end{itemize}

\textbf{Returns} Value at \(x\)

\textbf{Return type} float
class pySOT.optimization_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: \( f(0, 0, \ldots, 0) = 0 \)

Variables

- \texttt{dim} – Number of dimensions
- \texttt{lb} – Lower variable bounds
- \texttt{ub} – Upper variable bounds
- \texttt{int\_var} – Integer variables
- \texttt{cont\_var} – Continuous variables
- \texttt{min} – Global minimum value
- \texttt{minimum} – Global minimizer
- \texttt{info} – String with problem info

eval(x)

Evaluate the Sphere function at \( x \).

Parameters \( x \) (\texttt{numpy.array}) – Data point

Returns Value at \( x \)

Return type float

class pySOT.optimization_problems.SumOfSquares(dim=10)

Sum of squares function

\[ f(x_1, \ldots, x_n) = \sum_{i=1}^{n} i x_i^2 \]

Global optimum: \( f(0, 0, \ldots, 0) = 0 \)

Variables

- \texttt{dim} – Number of dimensions
- \texttt{lb} – Lower variable bounds
- \texttt{ub} – Upper variable bounds
- \texttt{int\_var} – Integer variables
- \texttt{cont\_var} – Continuous variables
- \texttt{min} – Global minimum value
- \texttt{minimum} – Global minimizer
- \texttt{info} – String with problem info

eval(x)

Evaluate the Sum of squares function at \( x \).
Parameters \( x (\text{numpy.array}) \) – Data point

Returns Value at \( x \)

Return type float

class pySOT.optimization_problems.Weierstrass (\text{dim} = 10)

eval (x)

Evaluate the Weierstrass function at \( x \).

Parameters \( x (\text{numpy.array}) \) – Data point

Returns Value at \( x \)

Return type float

class pySOT.optimization_problems.Zakharov (\text{dim} = 10)

Zakharov function

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

Variables

- \text{dim} – Number of dimensions
- \text{lb} – Lower variable bounds
- \text{ub} – Upper variable bounds
- \text{int\_var} – Integer variables
- \text{cont\_var} – Continuous variables
- \text{min} – Global minimum value
- \text{minimum} – Global minimizer
- \text{info} – String with problem info

eval (x)

Evaluate the Zakharov function at \( x \).

Parameters \( x (\text{numpy.array}) \) – Data point

Returns Value at \( x \)

Return type float

6.5 pySOT.strategy module

6.6 pySOT.surrogate module

6.7 pySOT.utils module

Module utils

Author David Eriksson <dme65@cornell.edu>
class pySOT.utils.GeneticAlgorithm(function, dim, lb, ub, int_var=None, pop_size=100, num_gen=100, start='SLHD')

Genetic algorithm.

Implementation of the real-valued Genetic algorithm. The mutations are normally distributed perturbations, the selection mechanism is a tournament selection, and the crossover operation is the standard linear combination taken at a randomly generated cutting point.

The total 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 pop_size x dim and return a numpy.array of size pop_size x 1
- dim (int) – Number of dimensions
- lb (numpy.array) – Lower variable bounds, of length dim
- ub (numpy.array) – Lower variable bounds, of length dim
- int_var (list) – List of indices with the integer valued variables (e.g., [0, 1, 5])
- pop_size (int) – Population size
- num_gen (int) – Number of generations
- start (string) – Method for generating the initial population

Variables

- nvariables – Number of variables (dimensions)
- 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 perturbation 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

optimize()

Method used to run the Genetic algorithm

Returns

Returns the best individual and its function value

Return type

numpy.array, float

pySOT.utils.check_opt_prob(obj)

Check an implementation of the optimization problem.

This method checks everything, but can’t make sure that the objective function returns 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.
Parameters **obj** (*Object*) – Optimization problem

:raise Appropriate error if object doesn’t follow the pySOT standard

```python
pySOT.utils.from_unit_box(x, lb, ub)
```

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
- **lb** (*numpy.array*) – Lower bounds, of size 1 x dim
- **ub** (*numpy.array*) – Upper bounds, of size 1 x dim

**Returns** Points mapped to the original domain

**Return type** *numpy.array*

```python
pySOT.utils.progress_plot(controller, title=", interactive=False)
```

Makes a progress plot from a POAP controller.

This method requires 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

```python
pySOT.utils.round_vars(x, int_var, lb, ub)
```

Round integer variables to closest integer in the domain.

**Parameters**

- **x** (*numpy.array*) – Set of points, of size npts x dim
- **int_var** (*numpy.array*) – Set of indices of integer variables
- **lb** (*numpy.array*) – Lower bounds, of size 1 x dim
- **ub** (*numpy.array*) – Upper bounds, of size 1 x dim

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

**Return type** *numpy.array*

```python
pySOT.utils.to_unit_box(x, lb, ub)
```

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
- **lb** (*numpy.array*) – Lower bounds, of size 1 x dim
- **ub** (*numpy.array*) – Upper bounds, of size 1 x dim

**Returns** Points mapped to the unit box

**Return type** *numpy.array*

```python
pySOT.utils.unit_rescale(x)
```

Shift and rescale elements of a vector to the unit interval

**Parameters** **x** (*numpy.ndarray*) – array that should be rescaled to the unit interval

**Returns** Array scaled to the unit interval
Return type  numpy.ndarray
CHAPTER 7

Changes

7.1 v.0.2.2, 2019-02-12

• Experimental designs can now map and round to domains
• Support for generating multiple experimental designs and picking the best

7.2 v.0.2.1, 2019-01-26

• Removing numpy.asmatrix calls, since this is now deprecated

7.3 v.0.2.0, 2018-12-06

• Most of the pySOT codebase has been rewritten
• We support asynchronous function evaluations
• The strategy has been merged with the adaptive sampling
• The penalty method strategy has been removed, but may be added back later
• A CheckpointController has been added that enables resuming terminated runs
• Python 2 support has been dropped, we now support Python 3.4 and later
• Expected improvement (EI) and lower confidence bound (LCB) have been added

7.4 v.0.1.36, 2017-07-20

• The GUI is now built in PyQt5 instead of PySide
7.5 v.0.1.35, 2017-04-29

- Added support for termination based on elapsed time
- Added the Hartman6 test problem

7.6 v.0.1.34, 2017-03-28

- Added support for adding points with known (and unknown) function values to the experimental design

7.7 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

7.8 v.0.1.32, 2016-12-07

- Switched to make py-earth, matlab_wrapper, and subprocess32 optional dependencies to resolve pip installation issues

7.9 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

7.10 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

**7.11 v0.1.29, 2016-10-20**

• Correcting an error in the pypi upload

**7.12 v0.1.28, 2016-10-20**

• Making the GUI work with the new RBF design

**7.13 v0.1.27, 2016-10-18**

• Removed dimensionality argument for the RBF to match the other surrogates

**7.14 v0.1.26, 2016-10-14**

• Significant 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

**7.15 v0.1.25, 2016-09-14**

• Fixed a bug in DYCORS when the subset has length 1

**7.16 v0.1.24, 2016-08-04**

• Changed to setup.py to use rst format for pypi

**7.17 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
7.18 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

7.19 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

7.20 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

7.21 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

7.22 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.

7.23 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
7.24  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

7.25  v0.1.16, 2016-01-06

- Added a projection strategy

7.26  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

7.27  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.

7.28  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.

7.29  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

7.30  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

7.31 v0.1.10, 2015-07-14
- README.md not uploaded to pypi which caused pip install to fail

7.32 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

7.33 v0.1.8, 2015-07-01
- Multi Start Gradient method that uses the L-BFGS-B algorithm to search on the surrogate

7.34 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

7.35 v0.1.6, 2015-06-28
- GUI inactivates all buttons but the stop button while running
- Bug fixes

7.36 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
7.37 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

7.38 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)

7.39 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

7.40 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

7.41 v0.1.0, 2015-06-03

- Initial release
Copyright (c) 2015 by David Bindel, David Eriksson, and contributors. See Contributors for more details.

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
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

a
auxiliary_problems, 21

c
controller, 25

e
experimental_design, 26

o
optimization_problems, 28

p
pySOT.auxiliary_problems, 21
pySOT.controller, 25
pySOT.experimental_design, 26
pySOT.optimization_problems, 28
pySOT.utils, 37

u
utils, 37
Index

A
Ackley (class in pySOT.optimization_problems), 28
auxiliary_problems (module), 21

B
Branin (class in pySOT.optimization_problems), 28

C
candidate_dycors() (in module pySOT.auxiliary_problems), 21
candidate_srbf() (in module pySOT.auxiliary_problems), 22
candidate_uniform() (in module pySOT.auxiliary_problems), 22
check_opt_prob() (in module pySOT.utils), 38
CheckpointController (class in pySOT.controller), 25
candidate_dycors() (in module pySOT.experimental_design), 26
ExperimentalDesign (class in pySOT.experimental_design), 26

E
ei_merit() (in module pySOT.auxiliary_problems), 22
eval() (pySOT.optimization_problems.Ackley method), 28
eval() (pySOT.optimization_problems.Branin method), 29
eval() (pySOT.optimization_problems.Exponential method), 29
eval() (pySOT.optimization_problems.GoldsteinPrice method), 29
eval() (pySOT.optimization_problems.Griewank method), 30
eval() (pySOT.optimization_problems.Hartman3 method), 31
eval() (pySOT.optimization_problems.Hartman6 method), 31
eval() (pySOT.optimization_problems.Himmelblau method), 32
eval() (pySOT.optimization_problems.Levy method), 32
eval() (pySOT.optimization_problems.Michalewicz method), 33
eval() (pySOT.optimization_problems.OptimizationProblem method), 33
eval() (pySOT.optimization_problems.Perm method), 33
eval() (pySOT.optimization_problems.Rastrigin method), 34
eval() (pySOT.optimization_problems.Rosenbrock method), 34
eval() (pySOT.optimization_problems.Schwefel method), 35
eval() (pySOT.optimization_problems.SixHumpCamel method), 35
eval() (pySOT.optimization_problems.Sphere method), 36
eval() (pySOT.optimization_problems.SumOfSquares method), 36
expected_improvement_ga() (in module pySOT.auxiliary_problems), 23
expected_improvement_uniform() (in module pySOT.auxiliary_problems), 23
ExperimentalDesign (class in pySOT.experimental_design), 26

F
from_unit_box() (in module pySOT.utils), 39

G
generate_points() (pySOT.experimental_design.ExperimentalDesign method), 26
generate_points() (pySOT.experimental_design.LatinHypercube method), 26
generate_points() (pySOT.experimental_design.SymmetricLatinHypercube method), 27
generate_points() (pySOT.experimental_design.TwoFactorial method), 27
GeneticAlgorithm (class in pySOT.utils), 37
GoldsteinPrice (class in pySOT.optimization_problems), 29
Griewank (class in pySOT.optimization_problems), 30
H
Hartman3 (class in pySOT.optimization_problems), 30
Hartman6 (class in pySOT.optimization_problems), 31
Himmelblau (class in pySOT.optimization_problems), 31
L
LatinHypercube (class in pySOT.experimental_design), 26
lcb_merit() (in module pySOT.auxiliary_problems), 24
Levy (class in pySOT.optimization_problems), 32
lower_confidence_bound_ga() (in module pySOT.auxiliary_problems), 24
M
Michalewicz (class in pySOT.optimization_problems), 32
O
on_cancel() (pySOT.controller.CheckpointController method), 25
on_complete() (pySOT.controller.CheckpointController method), 25
on_kill() (pySOT.controller.CheckpointController method), 25
on_new_feval() (pySOT.controller.CheckpointController method), 25
on_terminate() (pySOT.controller.CheckpointController method), 25
on_update() (pySOT.controller.CheckpointController method), 25
optimization_problems (module), 28
OptimizationProblem (class in pySOT.optimization_problems), 33
optimize() (pySOT.utils.GeneticAlgorithm method), 38
P
Perm (class in pySOT.optimization_problems), 33
progress_plot() (in module pySOT.utils), 39
pySOT.auxiliary_problems (module), 21
pySOT.controller (module), 25
pySOT.experimental_design (module), 26
pySOT.optimization_problems (module), 28
pySOT.utils (module), 37
R
Rastrigin (class in pySOT.optimization_problems), 33
resume() (pySOT.controller.CheckpointController method), 25
Rosenbrock (class in pySOT.optimization_problems), 34
round_vars() (in module pySOT.utils), 39un() (pySOT.controller.CheckpointController method), 26
S
Schwefel (class in pySOT.optimization_problems), 34
SixHumpCamel (class in pySOT.optimization_problems), 35
Sphere (class in pySOT.optimization_problems), 35
SumOfSquares (class in pySOT.optimization_problems), 36
SymmetricLatinHypercube (class in pySOT.experimental_design), 27
T
to_unit_box() (in module pySOT.utils), 39
TwoFactorial (class in pySOT.experimental_design), 27
U
unit_rescale() (in module pySOT.utils), 39
utils (module), 37
W
Weierstrass (class in pySOT.optimization_problems), 37
weighted_distance_merit() (in module pySOT.auxiliary_problems), 24
Z
Zakharov (class in pySOT.optimization_problems), 37