Surrogate Optimization Toolbox (pySOT) - 0.1.2 Tutorial

DAVID ERIKSSON
DAVID BINDEL
CHRISTINE SHOEMAKER

Cornell University
Center for Applied Mathematics
dme65@cornell.edu

24th June, 2015

Contents

1 Change history:

- (**0.1.2**)
  - 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 sub process test, to avoid platform dependency issues
- (**0.1.1**)
  - surrogate_optimizer.py was 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
- (**0.1.0**)
  - Initial release

2 Introduction

This is a tutorial (user guide) for the Surrogate Optimization Toolbox (pySOT) for global deterministic optimization problems. 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 are 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. The toolbox is based on the following published papers that should be cited when the toolbox is used for own research purposes:

1. J. Muller and R. Piche, 2011. "Mixture Surrogate Models Based on Dempster-Shafer Theory for Global Optimization Problems", Journal of Global Optimization, vol. 51, pp. 79-104
2. J. Muller, C.A. Shoemaker, and R. Piche, 2012. "SO-MI: A Surrogate Model Algorithm for Computationally Expensive Nonlinear Mixed-Integer Black-Box Global Optimization Problems", Computers & Operations Research, http://dx.doi.org/10.1016/j.cor.2012.08.022
3. R.G. Regis and C.A. Shoemaker, 2007. "A Stochastic Radial Basis Function Method for the Global Optimization of Expensive Functions", INFORMS Journal on Computing, vol. 19, pp. 497-509
4. R.G. Regis and C.A. Shoemaker, 2009. "Parallel Stochastic Global Optimization Using Radial Basis Functions", INFORMS Journal on Computing, vol. 21, pp. 411-426

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.

3 Licensing

Please refer to LICENSE.txt

4 Surrogate Model Algorithms

Surrogates models (or response surfaces) are 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.

1. Generate an initial experimental design.
2. Carry out the costly function evaluations at the points generated in Step 1.
3. Fit a response surface to the data generated in Steps 1 and 2.

4. Use the response surface to predict the objective function values at new points in the variable domain in order to decide the next point(s) to be evaluated.
5. Do the expensive function evaluation at the point(s) selected in Step 4.
6. Use the new data to update the surrogate model.
7. Iterate through Steps 4 to 6 until the stopping criterion has been met.

Surrogate model algorithms in the literature differ mainly with respect to

- The generation of the initial experimental design;
- The chosen surrogate model;
- The strategy for selecting the sample point(s) in each iteration.

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.

5 Installation

Before starting you will need Python 2.7 and pypi (pip). There are currently two ways to install the toolbox:

1. The easiest way to install the toolbox is through pypi in which case the following command should suffice (you may need sudo for UNIX):

```
pip install pySOT
```

2. (a) Clone the repository:

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

   or alternatively download the repository directly:

   i. Go to https://github.com/dme65/pySOT

   ii. Download the repository, extract the zip folder and change the name to pySOT

   (b) Navigate to the repository using:

   ```
   cd pySOT
   ```

   (c) Install dependencies:

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

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

   ```
   python setup.py install
   ```

(e) Several test problems are available at ./pySOT/test

Optional: If you want to use MARS you need to install the py-earth toolbox (http://github.com/jcrudy/py-earth)

6 Sphinx documentation

The necessary files to build the Sphinx documentation are provided in the docs subdirectory. We use the napoleon extension so you need to make sure you have this package. This can be done through pip

```
pip install sphinxcontrib-napoleon
```

To build the documentation run the command:

```
make html
```

7 Options

These are the the components and the supported options:

7.1 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. The following experimental designs are supported:

- **LatinHypercube**. Arguments:
  - **dim:** Number of dimensions
  - **npts:** Number of points to generate ($2\text{dim} + 1$ is recommended)

  Example:

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

  creates a Latin hypercube design with 10 points in 3 dimensions

- **SymmetricLatinHypercube** Arguments:
  - **dim:** Number of dimensions
  - **npts:** Number of points to generate ($2\text{dim} + 1$ is recommended)

  Example:

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

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

7.2 Surrogate model

The surrogate model approximates the underlying objective function given all of the points that have been evaluated. The following surrogate models are supported:

- **RBFInterpolant**. A radial basis function interpolant. Arguments:
  - **phi**: Kernel function. The options are
    - **phi_linear:** Linear RBF
    - **phi_cubic:** Cubic RBF
    - **phi_plate:** Thin-Plate RBF
  - **P:** Tail functions. The options are
    - **const_tail:** Constant tail
    - **linear_tail:** Linear tail
  - **dphi:** Derivative of kernel function. The options are:
    - **dphi_linear:** Derivative of Linear RBF
    - **dphi_cubic:** Derivative of Cubic RBF
    - **dphi_plate:** Derivative of Thin-Plate RBF
  - **dP:** Gradient of tail functions. The options are:
    - **dconst_tail:** Derivative of Constant tail
    - **dlinear_tail:** Derivative of Linear tail
  - **eta:** Regularization parameter. Default is 1e-8
  - **maxp:** Initial maximum number of points (can grow). Default is 100.

  Example:

  ```python
  from pySOT import RBFInterpolant, phi_cubic, linear_tail, \
                    dphi_cubic, dlinear_tail
  fhat = RBFInterpolant(phi=phi_cubic, P=linear_tail, dphi=dphi_cubic,
                        dP=dlinear_tail, eta=1e-8, maxp=500)
  ```

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

- **KrigingInterpolant:** A Kriging interpolant. Arguments:
  - **maxp:** Maximum number of points (can grow). Default is 100

Example:

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

creates a Kriging interpolant with a capacity of 500 points.

- **MARSInterpolant:** Generate a Multivariate Adaptive Regression Splines (MARS) model. Arguments:
  - **maxp:** Maximum number of points (can grow). Default is 100

  Example:

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

  creates a MARS interpolant with 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. The arguments are:
  - **model_list:** A list of surrogate model objects to be used.
  - **maxp:** Maximum number of points (can grow). Default is 100

  Example:

```
from pySOT import RBFInterpolant, phi_cubic, dphi_cubic, linear_tail, \
        dlinear_tail, MARSInterpolant

fhat1 = RBFInterpolant(phi_cubic, linear_tail,
                              dphi_cubic, dlinear_tail, 1e-8, 500)
fhat2 = MARSInterpolant(500)
models = [fhat1, fhat2]
fhat = EnsembleSurrogate(models, 500)
```

creates an ensemble surrogate with two surrogate models, namely a Cubic RBF and a MARS interpolant.

Note: The user is responsible for resetting the response surface after each experiment and this is done by calling the reset() method.

7.3 Capped RBF model

Functions with very large function values can cause the fitted surface to oscillate wildly. In the case of the RBFInterpolant we therefore provide a capped version that replaces every value above the median by the median value. This adapter takes an existing response surface and replaces it with a modified version in which any function values above the median are replaced by the median value.

Example:

```
from pySOT import RSCapped, RBFInterpolant, phi_cubic, linear_tail, \
                 dphi_cubic, dlinear_tail
fhat = RSCapped(RBFInterpolant(phi=phi_cubic, P=linear_tail, dphi=dphi_cubic,
                                dP=dlinear_tail, eta=1e-8, maxp=500))
```

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

7.4 Objective function

The objective function 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 must always be specified in the objective function class:

- **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 integer constraints, set to [ ]
- **continuous:** Specifies the continuous variables. If no variables are continuous, set to [ ]

The following methods must also exist.

- **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.

- **eval_ineq_constraints:** Only necessary if there are non-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 size $n \times M$ where M is the number of inequality constraints.

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 *validate* which is available in pySOT is helpful in order to test that the objective function is compatible with the framework.

7.5 Generation of next point to evaluate

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. We start by listing all 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 DDS 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
- **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, MultiSearchStrategy, CandidateDyCORS, \
                          CandidateDyCORS_CONT, CandidateDyCORS_INT, \
                          CandidateUniform

data = LinearMI()  # Optimization problem
search_strategies = [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)]
weights = [0, 1, 2, 3]
search_strategy = MultiSearchStrategy(search_strategies, weights)
```

8 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)
```

8.1 Controller

pySOT needs only the ThreadController, where we create a team of workers (threads) that carry our objective function evaluations. If the objective function is an external program we use workers of the class ProcessWorkerThread, whilst if the objective function isn't external we can just use the BasicWorkerThread class.

8.2 Strategies

pySOT provides two strategies:

- **SyncStrategyNoConstraints:** This strategy is to be used in case there are only bound constraints and no additional constraints. The arguments to this strategy are:
  - **worker_id:** An idea that the controller can use to distinguish between multiple simultaneously running optimization problems.

  - **data:** Objective function object, as described in Section 7.4
  - **response_surface:** Response surface object, as described in Section 7.2
  - **maxeval:** Maximum number of function evaluations
  - **nsamples:** Maximum number of simultaneous function evaluations (can be set to the number of workers/threads)
  - **exp_design:** Experimental design to do the initial evaluations, as described in Section 7.1. Default is a Latin Hypercube with 2dim+1 points
  - **search_procedure** Method to propose new evaluations, as described in Section 7.5. Default is Candidate DyCORS with 100dim candidate points.
  - **extra:** Additional point to be added to the experimental design. If a good solution is known, you can use this argument to make sure this point is evaluated early.
  - **quiet:** Set to false if you want all messages to be suppressed. Default is True and the value of each evaluation is printed when it finishes.
  - stream: If you want pySOT to print messages during the optimization run to some other place than stdout you can provide a stream here. If no argument is given, messages are printed to stdout.

- **SyncStrategyPenalty:** If there are additional non-bound constraints we provide a penalty based strategy. This strategy assumes that it makes sense to evaluate the objective function outside the feasible region. The strategy also assumes that there is a method eval_ineq_constraints that works exactly as described in Section 7.4. The startegy takes the same argument as SyncStrategyNoConstraints plus one addition argument which is the penalty to be used in the penalty method. Given a penalty μ set by the user we try to solve the box-constrained optimization problem

$$\begin{array}{ll} \underset{x}{\text{minimize}} & \widetilde{f}(x) = f(x) + \mu \sum_{i=1}^{M} \max(0, g_i(x))^2 \\ \text{subject to:} & -\infty < \ell_i \leq x_i \leq u_i < \infty, \quad i = 1, \dots, n \end{array}$$

  where $x \in \mathbb{R}^n$ and there are M inequality constraints of the form $g_i(x) \leq 0$, for $i = 1, \dots, M$. If you want the resulting solution to be feasible, just set μ to a very large value. This will force the algorithms to work there way towards a feasible solution. Candidate points are generated based on the solution with the smallest value of $\widetilde{f}$. In order to rank function value prediction by the response surface we set all infeasible solutions to have the same prediction as the worst feasible candidate point. The reason for this is that large penalties make it impossible for the weighted distance criteria to distinguish between feasible points. This modified approach will make the algorithm prefer feasible candidate points over infeasible candidate points as long as the function value is weighted higher than the minimum distance.

9 Examples

This section provides several examples that shows how to use POAP and pySOT to solve optimization problems.

9.1 First example (Hello World)

This example is a continuous optimization problem of the 10-dimensional Ackley-function which has only bound constraints. We are using 4 threads and 1000 evaluations. To generate new points to evaluate we use Candidate DyCORS. The experimental design is created using a Latin Hypercube with 2dim+1 points. The response surface is a cubic radial basis function with a linear tail. We use the ThreadController and the SyncStrategyNoConstraints strategy, since there are only bound constraints. We launch 4 BasicWorkerThread, start the optimization and finally print the result to the screen.

```python
import logging
from pySOT import *
from poap.controller import ThreadController, BasicWorkerThread
import numpy as np
import os.path

if not os.path.exists("./logfiles"):
    os.makedirs("logfiles")
logging.basicConfig(filename="./logfiles/test_simple.log",
                    level=logging.INFO)
print("Number of threads: 4")
print("Maximum number of evaluations: 1000")
print("Search strategy: Candidate DyCORS")
print("Experimental design: Latin Hypercube")
print("Ensemble surrogates: Cubic RBF")

nthreads = 4
maxeval = 1000
nsamples = nthreads

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

# Create a strategy and a controller
controller = ThreadController()
controller.strategy = \
    SyncStrategyNoConstraints(
        worker_id=0, data=data,
        maxeval=maxeval, nsamples=nsamples,
        exp_design=LatinHypercube(dim=data.dim, npts=2*data.dim+1),
        response_surface=RBFInterpolant(phi=phi_cubic, P=linear_tail,
                                        dphi=dphi_cubic, dP=dlinear_tail,
                                        eta=1e-8, maxp=maxeval),
        search_procedure=CandidateDyCORS(data=data, numcand=200*data.dim))

# 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('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)))
```

9.2 Continuous problem with non-bound constraints

This example is a continuous optimization problem of the 10-dimensional Keane's bump function which has two non-bound constraints. We are using 4 threads and 500 evaluations. To generate new points to evaluate we use Candidate DyCORS. The experimental design is created using a Latin Hypercube with 2dim+1 points. The response surface is a cubic radial basis function with a linear tail. We use the ThreadController and the SyncStrategyPenalty strategy, with the default penalty 10^6. POAP doesn't check feasibility, so we need to modify the run command to supply a filter that makes POAP discard infeasible points when looking for the best solution at the end of the run. The feasible_merit method simply sets infeasible points to have infinite function value so that only feasible points are considered. If for example small constraint violations are accepted the user can provide a merit function that is more suitable such a case.

```
import logging
from pySOT import *
from poap.controller import ThreadController, BasicWorkerThread
import numpy as np
import os.path

if not os.path.exists("./logfiles"):
    os.makedirs("logfiles")
logging.basicConfig(filename="./logfiles/test_constraints.log",
                    level=logging.INFO)
print("Number of threads: 4")
print("Maximum number of evaluations: 500")
print("Search strategy: CandidateDycors")
print("Experimental design: Latin Hypercube")
print("Surrogate: Cubic RBF")

nthreads = 4
maxeval = 500
nsamples = nthreads

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

def feasible_merit(record):
    """Merit function for ordering final answers -- kill infeasible x"""
    x = record.params[0].reshape((1, record.params[0].shape[0]))
    if np.max(data.eval_ineq_constraints(x)) > 0:
        return np.inf
    return record.value

# Create a strategy and a controller
controller = ThreadController()
controller.strategy = \
    SyncStrategyPenalty(
        worker_id=0, data=data,
        maxeval=maxeval, nsamples=nsamples,
        response_surface=RBFInterpolant(phi=phi_cubic, P=linear_tail,
                                        dphi=dphi_cubic, dP=dlinear_tail,
                                        eta=1e-8, maxp=maxeval),
        exp_design=LatinHypercube(dim=data.dim, npts=2*data.dim+1),
        search_procedure=CandidateDyCORS(data=data, numcand=5000))

# Launch the threads
for _ in range(nthreads):
    worker = BasicWorkerThread(controller, data.objfunction)
    controller.launch_worker(worker)

result = controller.run(merit=feasible_merit)
best, xbest = result.value, result.params[0]

print('Best value: {0}'.format(best))
print('Best solution: {0}'.format(
    np.array_str(xbest, max_line_width=np.inf,
                 precision=5, suppress_small=True)))
```

9.3 Ensemble Surrogates

This example is a continuous optimization problem of the 3-dimensional Hartman3-function which has only bound constraints. We are using 4 threads and 50 evaluations. To generate new points to evaluate we use Candidate SRBF, since the optimization problem is low-dimensional. The experimental design is created using a Latin Hypercube with 2dim+1 points. We also add the extra point [0.1, 0.5, 0.8], which we pretend is a known good solution to the problem. The response surface is an ensemble surrogate consisting of a cubic radial basis function with a linear tail, a linear radial basis function with constant tail, a thin-plate radial basis function with linear tail, and a MARS interpolant. We also redirect the stream from stdout and let pySOT print all messages to the text file surrogate_optimizer.log placed in the current directory. We use the ThreadController and the SyncStrategyNoConstraints strategy, since there are only bound constraints. We launch 4 BasicWorkerThread, start the optimization and finally print the result to the screen.

```
import logging
from pySOT import *
from poap.controller import ThreadController, BasicWorkerThread
import numpy as np
import os.path

if not os.path.exists("./logfiles"):
    os.makedirs("logfiles")
```

```
logging.basicConfig(filename="./logfiles/test_ensemble.log",
                    level=logging.INFO)
print("Number of threads: 4")
print("Maximum number of evaluations: 50")
print("Search strategy: Candidate SRBF")
print("Experimental design: Latin Hypercube + point [0.1, 0.5, 0.8]")
print("Surrogate: Cubic RBF, Linear RBF, Thin-plate RBF, MARS")

nthreads = 4
maxeval = 50
nsamples = nthreads

data = Hartman3()
print(data.info)

# Use 3 differents RBF's and MARS as an ensemble surrogate
models = [
    RBFInterpolant(phi_cubic, linear_tail, dphi_cubic,
                   dlinear_tail, 1e-8, maxeval),
    RBFInterpolant(phi_linear, const_tail, dphi_linear,
                   dconst_tail, 1e-8, maxeval),
    RBFInterpolant(dphi_plate, linear_tail, dphi_plate,
                   dlinear_tail, 1e-8, maxeval),
]
response_surface = EnsembleSurrogate(models, maxeval)

# Add an additional point to the experimental design. If a good
# solution is already known you can add this point to the
# experimental design
extra = np.atleast_2d([0.1, 0.5, 0.8])

# Create a strategy and a controller
controller = ThreadController()
controller.strategy = \
    SyncStrategyNoConstraints(
        worker_id=0, data=data,
        response_surface=response_surface,
        maxeval=maxeval, nsamples=nsamples,
        exp_design=LatinHypercube(dim=data.dim, npts=2*data.dim+1),
        search_procedure=CandidateSRBF(data=data, numcand=200*data.dim),
        extra=extra)

# 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()

response_surface.compute_weights()
print('Final weights: {0}'.format(
    np.array_str(response_surface.weights, max_line_width=np.inf,
                 precision=5, suppress_small=True)))
```

```
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)))
```

9.4 Mixed-integer problem with non-bound constraints

This example is a mixed-integer optimization problem of a 5-dimensional problem with 3 inequality constraints. The first three variables are discrete and the last 2 are continuous. We are using 4 threads and 200 evaluations. To generate new points to evaluate we use a Multi-search strategy consisting of CandidateDyCORS, CandidateDyCORS_INT, CandidateDyCORS_CONT, and CandidateUniform. The experimental design is created using a Symmetric Latin Hypercube with 2dim+1 points. The response surface is a cubic radial basis function with a linear tail. We use the ThreadController and the SyncStrategyPenalty strategy, with the default penalty 10^6. We submit the same merit function as in the previous example with constraints. The best solution is finally printed to the screen.

```
import logging
from pySOT import *
from poap.controller import ThreadController, BasicWorkerThread
import numpy as np
import os.path

if not os.path.exists("./logfiles"):
    os.makedirs("logfiles")
logging.basicConfig(filename="./logfiles/test_mixed_integer_constraints.log",
                    level=logging.INFO)
print("Number of threads: 4")
print("Maximum number of evaluations: 200")
print("Search strategy: CandidateDyCORS, CandidateDyCORS_INT"
      ", CandidateDyCORS_CONT, CandidateUniform")
print("Experimental design: Symmetric Latin Hypercube")
print("Surrogate: Cubic RBF")

nthreads = 4
maxeval = 200
nsamples = nthreads

data = LinearMI()
print(data.info)

def feasible_merit(record):
    "Merit function for ordering final answers -- kill infeasible x"
    x = record.params[0].reshape((1, record.params[0].shape[0]))
    if np.max(data.eval_ineq_constraints(x)) > 0:
        return np.inf
    return record.value

exp_design = SymmetricLatinHypercube(dim=data.dim, npts=2*data.dim+1)
response_surface = RBFInterpolant(phi=phi_cubic, P=linear_tail,
                                  dphi=dphi_cubic, dP=dlinear_tail,
                                    eta=1e-8, maxp=maxeval)

# Use a multi-search strategy for candidate points
search_proc = MultiSearchStrategy(
    [CandidateDyCORS(data=data, numcand=200*data.dim),
     CandidateUniform(data=data, numcand=200*data.dim),
     CandidateDyCORS_INT(data=data, numcand=200*data.dim),
     CandidateDyCORS_CONT(data=data, numcand=200*data.dim)],
    [0, 1, 2, 3])

# Create a strategy and a controller
controller = ThreadController()
controller.strategy = \
    SyncStrategyPenalty(
        worker_id=0, data=data,
        response_surface=response_surface,
        maxeval=maxeval, nsamples=nsamples,
        exp_design=exp_design,
        search_procedure=search_proc)

# Launch the threads
for _ in range(nthreads):
    worker = BasicWorkerThread(controller, data.objfunction)
    controller.launch_worker(worker)

result = controller.run(merit=feasible_merit)
best, xbest = result.value, result.params[0]

print('Best value: {0}'.format(best))
print('Best solution: {0}'.format(
    np.array_str(xbest, max_line_width=np.inf,
                 precision=5, suppress_small=True)))
```

9.5 External C++ objective function

This last example shows how to use POAP and pySOT with an external objective function. We consider an objective function written in C++ that computes the sum of square of a given input, which is an easy convex optimization problem. The program takes its input as a string $'x_1, x_2, \ldots, x_n'$. With a probability of 0.9 the program prints the value of the sum of squares to the screen. With probability 0.1 the program prints nothing and terminates, which is supposed to imitate that the evaluation crashed. The C++ program that is compiled with the name sphere_ext is provided below:

```
#include <iostream>
#include <vector>
#include <sstream>
#include <unistd.h>
#include <numeric>
#include <random>

int main(int argc, char** argv) {
        // Random number generator
        std::random_device rand_dev;
        std::mt19937 generator(rand_dev());
        std::uniform_real_distribution<float>  distr(0.0, 1.0);

        // Pretend the simulation crashes with probability 0.1
        if(distr(generator) > 0.1) {

                // Convert input to a standard vector
                std::vector<float> vect;
                std::stringstream ss(argv[1]);
                float f;

                while (ss >> f) {
                        vect.push_back(f);
                        if (ss.peek() == ',')
                                ss.ignore();
                }
                printf("%g\n", std::inner_product(vect.begin(), vect.end(),
                                                  vect.begin(), 0.0 ));
        }
        return 0;
}
```

We will use the subprocess library in Python to launch the objective function evaluations to our compiled C++ program. The help method array2str converts a numpy array to a string '$x_1, x_2, \ldots, x_n$', which is what our C++ program wants as an input. The class SphereExt is the basic objective function class and the DummySim class overloads the evaluation method for a ProcessWorkerThread. In case the objective function evaluation fails, a message is printed to the screen just to illustrate that things are working correctly.

```python
import logging
from pySOT import *
from poap.controller import ThreadController, ProcessWorkerThread
import numpy as np
from subprocess import Popen, PIPE
import os.path

def array2str(x):
    return ",".join(np.char.mod('%f', x))


class SphereExt:
    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 Sphere function \n" + \
                    "Global optimum: f(0,0,...,0) = 0"
        self.min = 0
        self.integer = []
        self.continuous = np.arange(0, dim)


class DummySim(ProcessWorkerThread):

    def handle_eval(self, record):
        self.process = Popen(['./sphere_ext', array2str(record.params[0])],
                             stdout=PIPE)
        out = self.process.communicate()[0]
        try:
            val = float(out)  # This raises ValueError if out is not a float
            self.finish_success(record, val)
        except ValueError:
            logging.warning("Function evaluation crashed/failed")
            self.finish_failure(record)

if not os.path.exists("./logfiles"):
    os.makedirs("logfiles")
logging.basicConfig(filename="./logfiles/test_subprocess.log",
                    level=logging.INFO)

print("Number of threads: 4")
print("Maximum number of evaluations: 200")
print("Search strategy: Candidate DyCORS")
print("Experimental design: Latin Hypercube")
print("Ensemble surrogates: Cubic RBF")

assert os.path.isfile("./sphere_ext"), "You need to build sphere_ext"
nthreads = 4
maxeval = 200
nsamples = nthreads

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

# Create a strategy and a controller
controller = ThreadController()
controller.strategy = \
    SyncStrategyNoConstraints(
        worker_id=0, data=data,
        maxeval=maxeval, nsamples=nsamples,
        exp_design=LatinHypercube(dim=data.dim, npts=2*data.dim+1),
        search_procedure=CandidateDyCORS(data=data, numcand=200*data.dim),
        response_surface=RBFInterpolant(phi=phi_cubic, P=linear_tail,
                                        dphi=dphi_cubic, dP=dlinear_tail,
                                        eta=1e-8, maxp=maxeval))

# Launch the threads and give them access to the objective function
for _ in range(nthreads):
    controller.launch_worker(DummySim(controller))

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

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)))
```

10 Hierarchy of POAP + pySOT

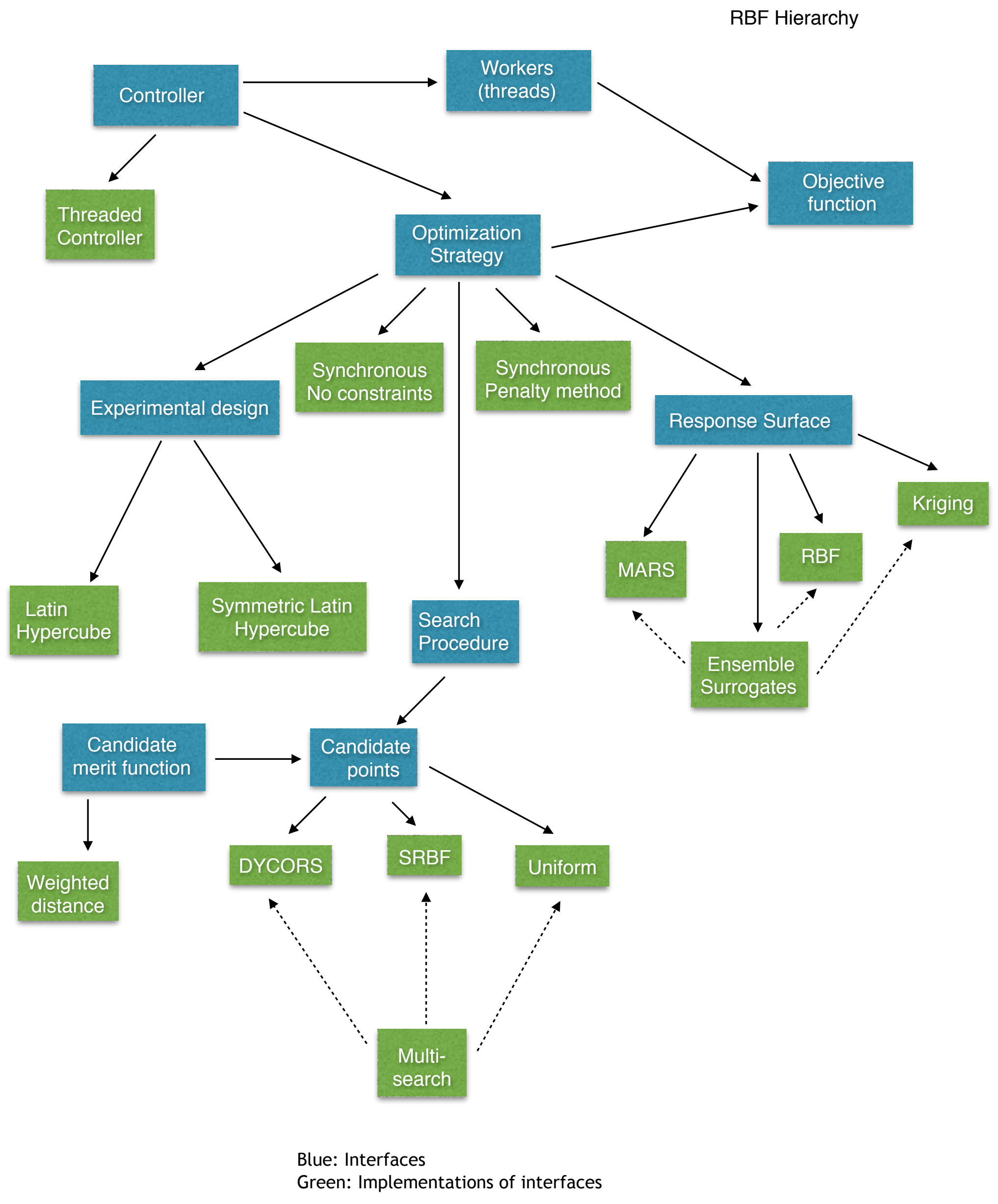

Figure 1: Overview of the pySOT hierarchy

11 Future changes

- Add an asynchronous strategy
- Add Heuristic Algorithms to search on the surrogate

- Add more experimental designs
- Add more methods for handling constraints, especially a barrier method
- Add a Graphical User Interface (GUI)
- Support for Python 3.x