Surrogate Optimization Toolbox (pySOT) - 0.1.20 Tutorial

DAVID ERIKSSON

DAVID BINDEL

CHRISTINE SHOEMAKER

Cornell University
Center for Applied Mathematics
dme65@cornell.edu

19th June, 2016

Contents

1 Change history:

- **(0.1.20)**

  - 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

- **(0.1.19)**

  - Too much regularization was added to the RBF surface when the volume of the domain was large.

- **(0.1.18)**

  - 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

- **(0.1.17)**

  - 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

- **(0.1.16)**

  - Added a projection strategy

- **(0.1.15)**

  - 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

- **(0.1.14)**

  - 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.

- **(0.1.13)**

- Allowed to still import the rest of pySOT when PySide is not found. In this case, the GUI will be unavailable.

- **(0.1.12)**

  - 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
  - DYCORS uses the remaining evaluation budget to change the probabilities after a restart instead of using the total budget

- **(0.1.11)**

  - 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 is 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

- **(0.1.10)**

  - README.md not uploaded to pypi which caused the pip install to fail

- **(0.1.9)**

  - 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

- **(0.1.8)**

  - Multi Start Gradient method that uses the L-BFGS-B algorithm to search on the surroagate

- **(0.1.7)**

  - 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

- **(0.1.6)**

- – GUI inactivates all buttons but the stop button while running
- – Bug fixes

- **(0.1.5)**

  - – 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

- **(0.1.4)**

  - – 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

- **(0.1.3)**

  - – 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, Sphere-Ext.py)

- **(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 take to evaluate, the more efficient are these algorithms. For this reason, this toolbox may not be very efficient for problems with computationally cheap function evaluations. Surrogate models are intended to be used when function evaluations take from several minutes to several hours or more. The toolbox is based on the following published papers that should be cited when the toolbox is used for own research purposes:

1. Stochastic RBF: Regis and Shoemaker [2007]

2. Parallel Stochastic RBF: Regis and Shoemaker [2009]

3. DYCORS: Regis and Shoemaker [2013]

4. Surrogate optimization using mixture surrogates: Müller and Piché [2011]

5. Surrogate optimization for Mixed-Integer problems: Müller et al. [2013]

6. Surrogate optimization for Integer problems: Müller et al. [2014]

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. The general framework for a Surrogate Optimization algorithm can be seen in Algorithm 1

Input: Initial experimental design, Sample point strategy, Surrogate model, Stopping criterion, Restart criterion

Output: Best solution and its corresponding function value

1 Generate an initial experimental design;
2 Evaluate the points in the experimental design;
3 Build a Surrogate model from the data;
4 **repeat**
5   **if** *Restart criterion met* **then**
6   Reset the Surrogate model and the Sample point strategy;
7   **go to** 1;
8   **end**
9   Use the Sample point strategy to generate new point(s) to evaluate;
10   Evaluate the point(s) generated using all computational resources;
11   Update the Surrogate model;
12 **until** *Stopping criterion met*;

Algorithm 1: Synchronous Surrogate Optimization Algorithm

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.x and pip. You must also have numpy and scipy installed and we recommend installing the Anaconda package (https://www.continuum.io/downloads).

Next you need to decide what optional packages you want:

- If you want to use MARS you need to install the py-earth toolbox (http://github.com/jcrudy/py-earth). It is possible to install py-earth with pip:

```
pip install https://github.com/jcrudy/py-earth/archive/master.zip
```

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

```
pip install PySide
```

There are currently two ways to install pySOT.

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

```
pip install pySOT
```

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

```
pip install inspyred
python setup.py install
```

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

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. Any implementation must have the following method

- generate_points(): Returns an experimental design of size $npts \times d$ where npts is the number of points in the initial design, which was specified when the object was created

The following experimental designs are supported:

- **LatinHypercube**. Arguments:

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

  Example:

  ```python
  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

- **TwoFactorial**. The corners of the unit hypercube. Arguments:

  - **dim:** Number of dimensions

  Example:

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

  creates a symmetric Latin hypercube design with 2^3 points in 3 dimensions

- **BoxBehnken**. Box-Behnken design with one center point. Arguments:

  - **dim:** Number of dimensions

Example:

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

creates a symmetric BoxBehnken design with 13 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. Any implementation of a surrogate model must have the following attributes and methods

Attributes:

- nump: Number of data points (integer)
- maxp: Maximum number of data points (integer)

Required methods:

- reset(): Resets the surrogate model
- get_x(): Returns a numpy array of size $nump \times d$ of the data points
- get_fx(): Returns a numpy array of length $nump$ with the function values
- add_point(x, f): Adds a point x with value f to the surrogate model
- eval(x): Evaluates the surrogate model at one point x
- evals(x): Evaluates the surrogate model at multiple points

Optional methods:

- deriv(x): Returns a numpy array with the gradient at one point x

The following surrogate models are supported:

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

    - **surftype**: Kernel function. The options are
        * **LinearRBFSurface:** Linear RBF (comes with a constant tail)
        * **CubicRBFSurface:** Cubic RBF (comes with a linear tail)
        * **TPSSurface:** Thin-Plate RBF (comes with a linear tail)
    - **maxp:** Initial maximum number of points (can grow). Default is 100.

    Example:

    ```python
    from pySOT import RBFInterpolant, CubicRBFSurface
    fhat = RBFInterpolant(surftype=CubicRBFSurface, maxp=500)
    ```

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

    Note: The RBF surfaces automatically applies damping to the RBF system in order to keep the system well-conditioned.

- **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, CubicRBFSurface, LinearRBFSurface, \
                  TPSSurface, EnsembleSurrogate

models = [
        RBFInterpolant(surftype=CubicRBFSurface, maxp=500),
        RBFInterpolant(surftype=LinearRBFSurface, maxp=500),
```

```
        RBFInterpolant(surftype=TPSSurface, maxp=500)
]

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

creates an ensemble surrogate with three surrogate models, namely a Cubic RBF Interpolant, a Linear RBF Interpolant, and a TPS RBF 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 transforms the function values. The default is to replace all function values larger than the median of the function values by the median, but it is possible to provide an other transformation. Arguments:

- **model**: Surrogate model.

- **transformation:** Function value transformation

Example:

```
from pySOT import RSCapped, RBFInterpolant, CubicRBFSurface

# Set inf and nan to the largest value observed so far
def transform(fvalues):
        ind = np.isfinite(fvalues)
        fvalues[np.logical_not(ind)] = np.max(fvalues[ind])
        return fvalues

fhat = RSCapped(RBFInterpolant(model=CubicRBFSurface, maxp=500), \
                transformation=transform)
```

creates a cubic RBF with a linear tail with a capacity for 500 points with capping that transforms inf and nan to the largest finite function value found so far.

7.4 Scaled RBF model

If the domain is large the linear system of equations for the RBF coefficients gets very ill-conditioned. One way of dealing with this problem is to rescale to domain to the unit hypercube, but the sampling methods may want to know exactly what the domain looks like. For this purpose we provide a wrapper for the surrogate model that internally rescales the domain to the unit hypercube, which keeps the interpolant stable while not affecting the other objects. Arguments:

- **model**: Surrogate model.

- **data:** Optimization problem object

Example:

```python
from pySOT import RSUnitBox, RBFInterpolant, CubicRBFSurface

# data is the optimization problem
fhat = RSUnitBox(RBFInterpolant(model=CubicRBFSurface, maxp=500), \
                 data=data)
```

creates a cubic RBF with a linear tail with a capacity for 500 points with capping that transforms inf and nan to the largest finite function value found so far.

7.5 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.

```python
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.

7.6 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. Each implementation of this object is required to have the following attributes and methods

Required attributes:

– *proposed_points*: Numpy array of the points that have been proposed for evaluation

Required methods:

– *init(start_sample, fhat, budget)*: This initializes the sampling strategy by providing the points that were evaluated in the experimental design phase, the response surface, and also provides the evaluation budget.

– *remove_point(x)*: Removes point x from list of proposed_points if the evaluation crashed or was never carried out by the strategy. Returns True if the point was removed and False if the removal failed.

– *make_points(npts, xbest, sigma, subset=None, proj_fun=None)*: This is the method that proposes *npts* new evaluations to the strategy. It needs to

know the number of points to propose, the best data point evaluated so far, the preferred sample radius of the strategy (w.r.t the unit box), the coordinates that the strategy wants to perturb, and a way to project points onto the feasible region.

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

- **CandidateSRBF:** Generate perturbations around the best solution found so far

- **CandidateSRBF_INT:** Uses CandidateSRBF but only perturbs the integer variables

- **CandidateSRBF_CONT:** Uses CandidateSRBF but only perturbs the continuous variables

- **CandidateDYCORS** Uses a DYCORS strategy which perturbs each coordinate with some iteration dependent probability. This probability is a monotonically decreasing function with the number of iteration.

- **CandidateDYCORS_CONT:** Uses CandidateDYCORS but only perturbs the continuous variables

- **CandidateDYCORS_INT:** Uses CandidateDYCORS but only perturbs the integer variables

- **CandidateDDS**: Uses the DDS strategy where only a few candidate points are generated and the one with the best surrogate prediction is picked for evaluation

- **CandidateDDS_CONT:** Uses CandidateDDS but only perturbs the continuous variables

- **CandidateDDS_INT:** Uses CandidateDDS but only perturbs the integer variables

- **CandidateUniform:** Chooses a new point uniformly from the box-constrained domain

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

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

The CandidateDYCORS algorithm is the bread-and-butter algorithm for any problems with more than 5 dimensions whilst CandidateSRBF is recommended for problems with only a few dimensions. It is sometimes efficient in mixed-integer problems to perturb the integer and continuous variables separately and we therefore provide such method for each of these algorithms. Finally, uniformly choosing a new point has the advantage of creating diversity to avoid getting stuck in

a local minima. Each method needs an objective function object as described in the previous section (the input name is data) and how many perturbations should be generated around the best solution found so far (the input name is numcand). Around 100 points per dimension, but no more than 5000, is recommended. Next is an example on how to generate a multi-start strategy that uses CandidateDY-CORS, 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.

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

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

8 POAP

pySOT uses POAP, which an event-driven framework for building and combining a synchronous 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)
```

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 7.5
  - **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 7.6. 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.

- **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 7.5. The strategy 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

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

  where $x \in \mathbb{R}^n$ and there are M inequality constraints of the form $g_i(x) \le 0$, for $i = 1, \ldots, M$. 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.

- **SyncStrategyProjection:** If there is an easy way to project infeasible points onto the feasible region we also provide a strategy that uses this projection operator in order to only evaluate feasible points. The user is responsible for making sure that the projection operator returns a feasible point. Candidate points are all projected onto the feasible region before the merit function is evaluated.

9   Guidelines for selecting parameters and components

Dimensions	Problem type	Search Strategies
≤ 10	Continuous	CandidateSRBF
> 10	Continuous	CandidateDYCORS
≤ 10	Integer	CandidateSRBF_INT
> 10	Integer	CandidateDYCORS_INT
≤ 10	Mixed	[CandidateSRBF, CandidateSRBF_INT, CandidateSRBF_CONT]
> 10	Mixed	[CandidateDYCORS, CandidateDYCORS_INT, CandidateDYCORS_CONT]

Non-bound constraints	Optimization Strategy
No	SyncStrategyNoConstraints
Yes	SyncStrategyPenalty or SyncStrategyProjection

Evaluation budget	Experimental design
< 10dim	Latin Hypercube with dim + 1 points
≥ 10dim	Symmetric Latin Hypercube with 2dim + 1 points

- **Response surface:** By default we recommend the CubicRBFSurface (with capping if necessary). We recommend not using Kriging since it is very slow.

- **Number of threads:** Setting both the number of simultaneous evaluations and the number of threads to the number of available cores.

10   Graphical user interface

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

```python
python
from pySOT import GUI
GUI()
```

or more compactly:

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

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

```python
import numpy as np

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

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

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

The four figures in Figure 1 show what the GUI looks like and how the optimization results are reported to the user. If the user want a search strategy that uses DYCORS, DYCORS, SRBF, DYCORS, DYCORS, SRBF, ... this can be achieved in the GUI by adding CandidateDYCORS twice and CandidateSRBF once.

11   Examples

The GitHub repository contains several examples for how to use pySOT.

12   Hierarchy of POAP + pySOT

(a) After initializing the GUI

(b) After importing an Optimization problem

(c) After choosing optimization components
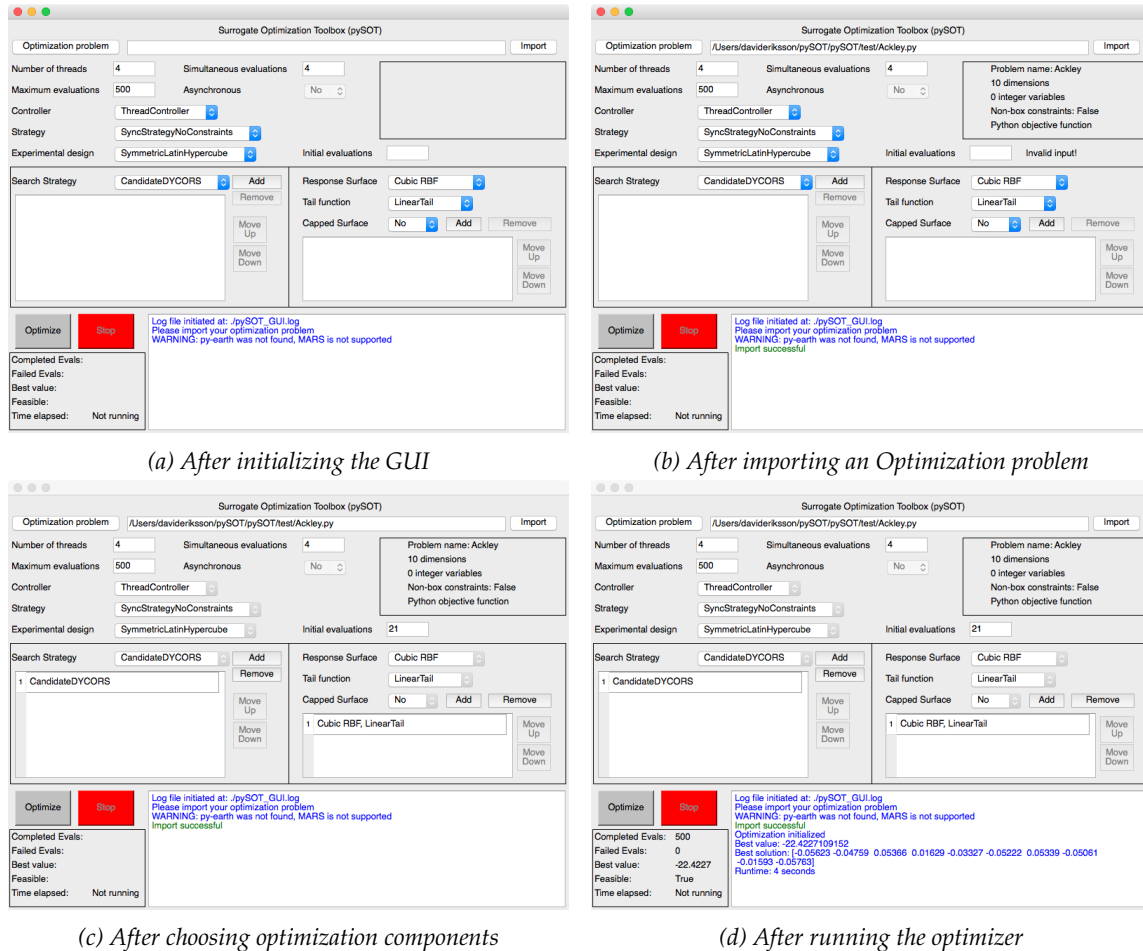
(d) After running the optimizer

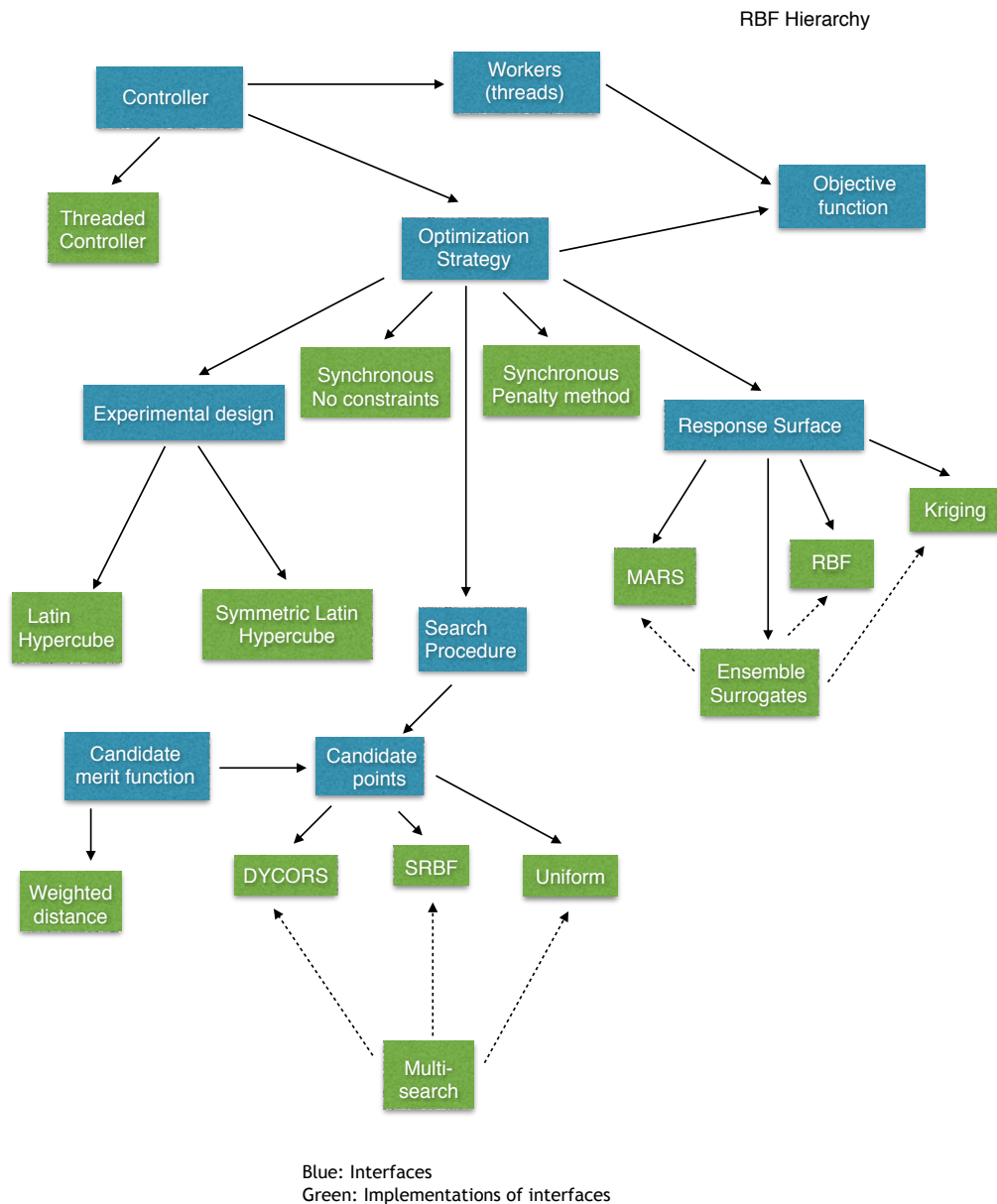*Figure 1: Illustration of the GUI*

Figure 2: Overview of the pySOT hierarchy

13 Future changes

- Add support for continuing from a logfile

- Add an asynchronous strategy

- Add more methods for handling constraints, especially a barrier method

- Support for Python 3.x

References

Rommel G Regis and Christine A Shoemaker. A stochastic radial basis function method for the global optimization of expensive functions. *INFORMS Journal on Computing*, 19(4):497–509, 2007.

Rommel G Regis and Christine A Shoemaker. Parallel stochastic global optimization using radial basis functions. *INFORMS Journal on Computing*, 21(3):411–426, 2009.

Rommel G Regis and Christine A Shoemaker. Combining radial basis function surrogates and dynamic coordinate search in high-dimensional expensive black-box optimization. *Engineering Optimization*, 45(5):529–555, 2013.

Juliane Müller and Robert Piché. Mixture surrogate models based on dempster-shafer theory for global optimization problems. *Journal of Global Optimization*, 51 (1):79–104, 2011.

Juliane Müller, Christine A Shoemaker, and Robert Piché. SO-MI: A surrogate model algorithm for computationally expensive nonlinear mixed-integer black-box global optimization problems. *Computers & Operations Research*, 40(5):1383–1400, 2013.

Juliane Müller, Christine A Shoemaker, and Robert Piché. SO-I: a surrogate model algorithm for expensive nonlinear integer programming problems including global optimization applications. *Journal of Global Optimization*, 59(4):865–889, 2014.