Surrogate Optimization Toolbox (pySOT) - 0.1.0 Tutorial

DAVID ERIKSSON

Cornell University
Center for Applied Mathematics
dme65@cornell.edu

4th June, 2015

Contents

1 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 Bitbucket (recommended) or to contact me at the email address: dme65@cornell.edu.

2 Licensing

Please refer to LICENSE.txt

3 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.

4 Installation

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

1. The easiest way to install the toolbox is to do it 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) Run a 30 dimensional test problem of the Ackley objective function with 4 threads and 1000 evaluations:

```
python ./pySOT/test/test.py
```

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

5 Options

These are the the components and the supported options:

5.1 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
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:

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

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

5.2 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:

```
from pySOT import RBFInterpolant, phi_cubic, linear_tail, \
                  dphi_cubic, dlinear_tail
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:
  - **initp:** Number of initial points for setting up Kriging model. Use the same number of points that you generate from the experimental design. The Kriging model is not used before all points in the experimental design have been evaluated.
  - **maxp:** Maximum number of points (can grow). Default is 100

Example:

```
from pySOT import KrigingInterpolant
KrigingInterpolant(initp=20, maxp=500)
```

generates a Kriging interpolant that needs 20 initial points and 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
MARSInterpolant(maxp=500)
```

generates a MARS interpolant with a capacity of 500 points.

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

5.3 Capped RBF model

Functions with exponential behaviors can cause the fitted surface to oscillate wildly. In the case of the RBFInterpolant we therefore provide a capping strategy 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
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.

5.4 Constraint Method

In order to handle general inequality constraints we have implemented a simple penalty method that starts with a penalty μ set by the user and then solves the box-constrained optimization problem

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

where $x \in \mathbb{R}^n$ and there are M inequality constraints of the form $g_i(x) \leq 0$, for $i = 1, \ldots, M$. If the best solution found is infeasible the penalty is multiplied by 10 and the algorithm restarts. This is carried out until the best solution found is feasible.

Example:

```
from pySOT import PenaltyMethod
PenaltyMethod(penalty=1.0)
```

Note: The user is responsible for resetting the constraint handler after each experiment and this is done by calling the reset() method. The constraint handler is not reset to make the penalty that gave a feasible solution accessible for the user, so that this value can guide the penalty for the next experiment.

5.5 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 [ ]
- **constraints:** Set to True if there are inequality constraints in addition to the bound constraints. Set to False otherwise

The following methods must also exist.

- **objfunction:** Takes one input in the form of an numpy.ndarray, which corresponds to one point in dim dimensions. Returns the value of the objective function
- **eval_ineq_constraints:** Only necessary if the attribute constraints is true. Takes one input in the form of an numpy.ndarray, which corresponds to one point in dim dimensions. Returns a numpy.matrixlib.defmatrix.matrix of size $1 \times M$ where M is the number of inequality constraints that are not bound 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)
        self.constraints = True

    def eval_ineq_constraints(self, x):
        vec = np.zeros((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 in test_problems.py is helpful in order to test that the objective function is compatible with the framework.

5.6 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. 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
- **DyCORS** 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)
```

5.7 Surrogate optimizer

We provide a synchronous parallel framework for solving the optimization problem. The method of interest is called optimize and is available in *surrogate_optimizer.py*. It takes the following input arguments:

- **nthreads:** Number of threads (threads) to be used
- **maxeval:** Maximal number of function evaluations
- **nsample:** Maximal number of simultaneous function evaluations allowed. It is recommended to set this value to nthreads
- **response_surface:** A surrogate model (response surface)
- **experimental_design:** An experimental design to generate initial points from
- **search_strategies:** A strategy for generating new points to evaluate (can be MultiSearchStrategy)
- **constraint_handler:** Method for handling non-bound constraints

Example:

```
from pySOT import optimize
xbest, fbest = optimize(nthreads=nthreads, data=data, maxeval=maxeval,
                        nsample=nsample, response_surface=rsurface,
                        experimental_design=expdes, search_strategies=sstrat,
                        constraint_handler=chandle)
```

The method returns to values, xbest and fbest, which is the best solution found and its objective function value. All of the information from the run is currently printed to an external logfile, ./surrogate_optimization.log, but if the user want this information available in a structure I can make this possible as well.

6 An example of how to minimize the 30-dimensional Ackley function

```
import numpy as np

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

    def objfunction(self, x):
        if len(x) != self.dim:
            raise ValueError('Dimension mismatch')
        n = float(len(x))
        return -20.0*exp(-0.2*sqrt(sum(x**2)/n)) - \
            exp(sum(cos(2.0*pi*x))/n) + 20 + exp(1)
```

```
from pySOT import CandidateDyCORS,  LatinHypercube, PenaltyMethod, RSCapped, \
                  RBFInterpolant, phi_cubic, linear_tail, dphi_cubic, \
                  dlinear_tail, Ackley, optimize
import numpy as np

if __name__ == "__main__":
    nthreads = 4  # Number of threads
    maxeval = 1000  # Maximal number of function evaluations
    nsample = nthreads  # Maximal number of simultaneous evaluations
    data = Ackley(dim=30)  # Optimization problem
    ed = LatinHypercube(dim=data.dim, npts=2*data.dim+1)  # Experimental design
    rs = RSCapped(RBFInterpolant(phi=phi_cubic, P=linear_tail,
                                 dphi=dphi_cubic, dP=dlinear_tail,
                                 eta=1e-8, maxp=maxeval))  # Surrogate
    ss = CandidateDyCORS(data=data, numcand=100*data.dim)  # Search strategy
    ch = PenaltyMethod(penalty=1.0)  # Constraint handling

    print(data.info)
    # Optimize the objective function in synchronous parallel using nthreads
    # threads and at most maxeval function evaluations
    xbest, fbest = optimize(nthreads=nthreads, data=data, maxeval=maxeval,
                            nsample=nsample, response_surface=rs,
```

```
                        experimental_design=ed, search_strategies=ss,
                        constraint_handler=ch)
```

7 Future changes

- Add an asynchronous parallel optimizer
- Add Heuristic Algorithms to search on the surrogate
- Add more experimental designs
- Add more methods for handling constraints
- Add ensemble surrogates
- A Graphical User Interface (GUI)