pySOT Documentation

David Eriksson, David Bindel, Christine Shoemaker

Jun 23, 2020

User Documentation

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

The main purpose of the toolbox is for optimization of computationally expensive black-box objective functions with continuous and/or integer variables. All variables are assumed to have bound constraints in some form where none of the bounds are infinity. The tighter the bounds, the more efficient are the algorithms since it reduces the search region and increases the quality of the constructed surrogate. This toolbox may not be very efficient for problems with computationally cheap function evaluations. Surrogate models are intended to be used when function evaluations take from several minutes to several hours or more.

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

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

CHAPTER 1

Quickstart

1.1 Dependencies

Before starting you will need Python 3.4 or newer. You need to have numpy, scipy, and pip installed and we recommend installing Anaconda/Miniconda for your desired Python version.

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

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

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

   or

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

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

   ```
   pip install mpi4py
   ```

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

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

1.2 Installation

There are currently two ways to install pySOT:

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

```
pip install pySOT
```

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

2.1. Clone the repository:

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

2.2. Navigate to the repository using:

```
cd pySOT
```

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

```
python setup.py install
```

Several examples are available in ./pySOT/examples and ./pySOT/notebooks

CHAPTER 2

Surrogate optimization

Surrogate optimization algorithms generally consist of four components:

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

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

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

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

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

Outputs: Best solution and its corresponding function value

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

CHAPTER 3

Options

3.1 Strategy

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

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

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

The following strategies are currently supported:

3.1.1 SRBFStrategy

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

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

The main idea is to pick the new evaluations from a set of candidate points where each candidate point is generated as an N(0, sigma^2) distributed perturbation from the current best solution. The value of sigma is modified based on progress and follows the same logic as in many trust region methods; we increase sigma if we make a lot of progress (the surrogate is accurate) and decrease sigma when we aren't able to make progress (the surrogate model is inaccurate). More details about how sigma is updated is given in the original papers.

After generating the candidate points we predict their objective function value and compute the minimum distance to previously evaluated point. Let the candidate points be denoted by C and let the function value predictions be s(x_i) and the distance values be d(x_i), both rescaled through a linear transformation to the interval [0,1]. This is done to put the values on the same scale. The next point selected for evaluation is the candidate point x that minimizes the weighted-distance merit function:

$$\text{merit}(x) := ws(x) + (1-w)(1-d(x))$$

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

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

3.1.2 DYCORStrategy

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

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

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

The parameters are the same as in the SRBF strategy.

3.1.3 SOPStrategy

This is an implementation of the SOP strategy by Krityakierne, Akhtar and Shoemaker:

Tipaluck Krityakierne, Taimoor Akhtar and Christine A. Shoemaker.
SOP: parallel surrogate global optimization with Pareto center selection for computationally expensive single objective problems.
Journal of Global Optimization, 66(3): 417-437, 2016.

The core idea of SOP is to maintain a ranked archive of all previously evaluated points, as per non-dominated sorting between two objectives, i.e., i) Objective function value(minimize) and ii) Minimum distance from other evaluated points(maximize). A sub-archive of center points is subsequently maintained via selection from the ranked evaluated points. The number of points in the sub-archive of centers should be equal to (or greater than) the number of parallel threads. Candidate points are generated around each 'center point' via the DYCORS sampling strategy, i.e., an N(0, sigma^2) distributed perturbation of a subset of decision variables. A separate value of sigma is maintained for each center point, where sigma is decreased if no progress is registered in the bi-objective objective value and distance criterion trade-off. One point is selected for expensive evaluation from each set of candidate points, based on the surrogate approximation only. Hence the merit function is s(x), where s(x) is the surrogate prediction.

Exploration and exploitation are simultaneously achieved (in parallel) via the bi-objective ranking of previously evaluated points, and subsequent selection of these points as centers of DYCORS perturbations. Exploitation is achieved when the point with best objective value is the perturbation center, and the candidate around it with best surrogate value is selected as the new evaluation point. Exploration is achieved when the point with the maximum distance (max-min) from other evaluated points is selected as the perturbation center.

Parameters are the same as in SRBF strategy, but exclude weights, and include the following:

- ncenters: Number of center points for candidate search where one point is selected for evaluation per, each center.

3.1.4 EIStrategy

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

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

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

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

The optimization strategy terminates when the evaluatio budget has been exceeded or when the EI of the next point falls below some threshold, where the default threshold is 1e-6 * (max(fX) - min(fX)).

- **Parameters:**
  - max_evals: Evaluation budget (int)
  - opt_prob: Optimization problem object, must implement OptimizationProblem
  - exp_design: Experimental design object, must implement ExperimentalDesign

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

3.1.5 LCBStrategy

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

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

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

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

The optimization strategy terminates when the evaluatio budget has been exceeded or when the LCB of the next point falls below some threshold, where the default threshold is 1e-6 * (max(fX) - min(fX)).

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

3.2 Experimental design

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

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

The following experimental designs are supported:

3.2.1 LatinHypercube

A Latin hypercube design

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

Example:

```
from pySOT.experimental_design import LatinHypercube
lhd = LatinHypercube(dim=3, num_pts=10)
```

creates a Latin hypercube design with 10 points in 3 dimensions

3.2.2 SymmetricLatinHypercube

A symmetric Latin hypercube design

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

Example:

```
from pySOT.experimental_design import SymmetricLatinHypercube
slhd = SymmetricLatinHypercube(dim=3, num_pts=10)
```

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

3.2.3 TwoFactorial

The corners of the unit hypercube

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

Example:

```
from pySOT.experimental_design import TwoFactorial
two_factorial = TwoFactorial(dim=3)
```

creates a two factorial design with 8 points in 3 dimensions

3.3 Surrogate model

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

- **Attributes:**
    - dim: Number of dimensions
    - lb: Lower variable bounds
    - ub: Upper variable bounds
    - output_transformation: Transformation to apply to function values before fitting (for example median capping)
    - num_pts: Number of points in the surrogate model
    - X: Data points, of size num_pts x dim, currently incorporated in the model
    - _X: Data points scaled to the unit hypercube. We use these internally to for conditioning reasons
    - fX: Function values at the data points
    - updated: True if all information is incorporated in the model, else a new fit will be triggered
- **Required methods**
    - reset(): Resets the surrogate model
    - add_points(x, fx): Adds point(s) x with value(s) fx to the surrogate model. This **SHOULD NOT** trigger a new fit of the model.
    - predict(x): Evaluates the surrogate model at points x
    - predict_deriv(x): Evaluates the derivative of surrogate model at points x
- **Optional methods**
    - predict_std(x): Evaluates the uncertainty of the surrogate model at points x

The following surrogate models are supported:

3.3.1 RBFInterpolant

A radial basis function (RBF) takes the form:

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

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

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

where $P_{ij} = p_j(x_i)$ and $\Phi_{ij} = \phi(\|x_i - x_j\|)$ The regularization parameter η allows us to avoid problems with potential poor conditioning of the system. Consider using the SurrogateUnitBox wrapper or manually scaling the domain to the unit hypercube to avoid issues with the domain scaling.

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

- **Parameters:**
  - dim: Number of dimensions (int)
  - lb: Lower variable bounds (numpy.array)
  - ub: Upper variable bounds (numpy.array)
  - output_transformation: Transformation to apply to function values before fitting (callable)
  - kernel: RBF kernel object, must implement Kernel. Default: CubicKernel()
  - tail: RBF polynomial tail object, must implement Tail. Default: LinearTail(dim)
  - eta: Regularization parameter. Use something small like 1e-6 if the domain is [0, 1]^dim

Example:

```
from pySOT.surrogate import RBFInterpolant, CubicKernel, LinearTail
lb, ub = np.zeros(5), np.ones(5)  # Domain is [0, 1]^5
rbf = RBFInterpolant(dim=5, lb=lb, ub=ub, kernel=CubicKernel(),␣
↪tail=LinearTail(dim=dim))
```

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

Example:

```
from pySOT.surrogate import RBFInterpolant, CubicKernel, LinearTail, median_capping
lb, ub = np.zeros(5), np.ones(5)  # Domain is [0, 1]^5
rbf = RBFInterpolant(
    dim=5, lb=lb, ub=ub, output_transformation=median_capping, kernel=CubicKernel(),␣
↪tail=LinearTail(dim=5))
```

will apply median capping (replace values above median by the median) to the function values before fitting. This is useful for minimization problems where we do not want large values to influence the fit of the model.

3.3.2 GPRegressor

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

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

- lb: Lower variable bounds (numpy.array)
- ub: Upper variable bounds (numpy.array)
- output_transformation: Transformation to apply to function values before fitting (callable)
- gp: GPRegressor model in scikit-learn. Uses the SE/RBF/Gaussian kernel as a default if None is passed.
- n_restarts_optimizer: Number of restarts in hyperparamater fitting (int)

Example:

```
from pySOT.surrogate import GPRegressor
lb, ub = np.zeros(5), np.ones(5)  # Domain is [0, 1]^5
gp = GPRegressor(dim=5, lb=lb, ub=ub)
```

creates a GPRegressor object in dim dimensions.

3.3.3 MARSInterpolant

Generate a Multivariate Adaptive Regression Splines (MARS) model.

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

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

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

- **Parameters:**
  - dim: Number of dimensions (int)
  - lb: Lower variable bounds (numpy.array)
  - ub: Upper variable bounds (numpy.array)
  - output_transformation: Transformation to apply to function values before fitting (callable)

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

Example:

```
from pySOT.surrogate import MARSInterpolant
lb, ub = np.zeros(5), np.ones(5)  # Domain is [0, 1]^5
mars = MARSInterpolant(dim=5, lb=lb, ub=ub)
```

creates a MARS interpolant in dim dimensions.

3.3.4 PolyRegressor

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

- **Parameters:**
  - dim: Number of dimensions (int)
  - lb: Lower variable bounds (numpy.array)
  - ub: Upper variable bounds (numpy.array)
  - output_transformation: Transformation to apply to function values before fitting (callable)
  - degree: Polynomial degree (int)

Example:

```
from pySOT.surrogate import PolyRegressor
lb, ub = np.zeros(5), np.ones(5)  # Domain is [0, 1]^5
poly = PolyRegressor(dim=5, lb=lb, ub=ub, degree=2)
```

creates a polynomial regressor of degree 2.

3.4 Optimization problem

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

- **Attributes**
  - lb: Lower bounds for the variables.
  - ub: Upper bounds for the variables.
  - dim: Number of dimensions
  - int_var: Specifies the integer variables. If no variables have discrete, set to []
  - cont_var: Specifies the continuous variables. If no variables are continuous, set to []
- **Required methods**
  - eval: Takes one input in the form of 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.

CHAPTER 4

POAP

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

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

# samples = list of sample points ...

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

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

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

4.1 Controller

The controller is responsible for accepting or rejecting proposals by the strategy object, controlling and monitoring the workers, and informing the strategy object of relevant events. Examples of relevant events are the processing of a proposal, or status updates on a function evaluation. Interactions between controller and the strategies are organized around proposals and evaluation records. At the beginning of the optimization and on any later change to the system state, the controller requests a proposal from the strategy. The proposal consists of an action (evaluate a function, kill a function, or terminate the optimization), a list of parameters, and a list of callback functions to be executed once the proposal is processed. The controller then either accepts the proposal (and sends a command to the worker), or rejects the proposal.

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

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

4.2 Strategy

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

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

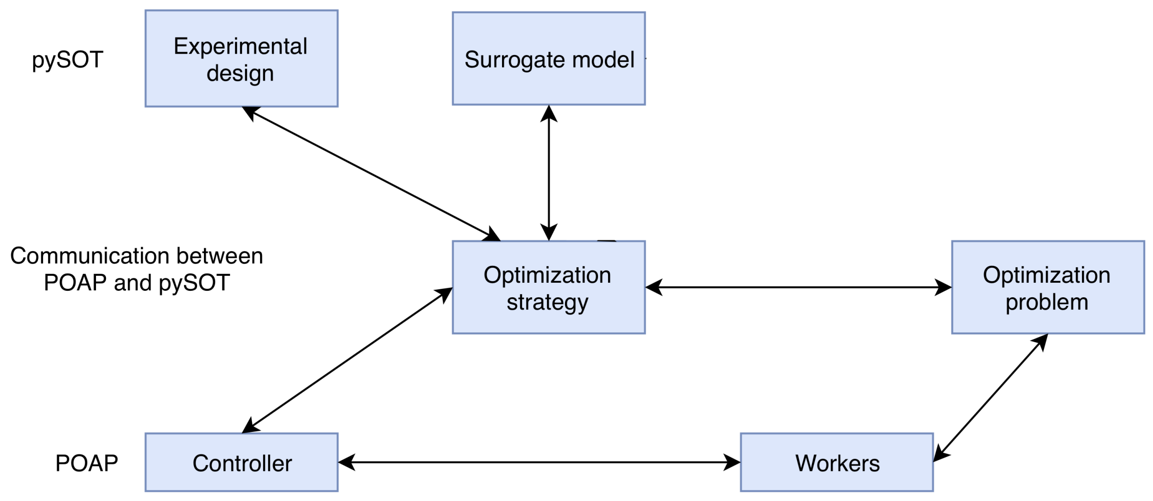
4.3 Workers

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

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

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

4.4 Communication between POAP and pySOT



CHAPTER 5

Logging

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

- critical
- error
- warning
- info
- debug

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

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

CHAPTER 6

Source code

6.1 pySOT.auxiliary_problems module

6.2 pySOT.controller module

6.3 pySOT.experimental_design module

6.4 pySOT.optimization_problems module

6.5 pySOT.strategy module

6.6 pySOT.surrogate module

6.7 pySOT.utils module

CHAPTER 7

Changes

7.1 v.0.3.3, 2020-06-22

- Exclude examples and tests from the package

7.2 v.0.3.2, 2020-06-18

- Fix import error

7.3 v.0.3.1, 2020-06-17

- Automatically scale domain to the unit hypercube for the surrogates
- Add output transformations (like median capping) to the surrogates

7.4 v.0.3.0, 2020-06-16

- Refactoring the code and fixing the docs
- Adding pre-commit hooks

7.5 v.0.2.3, 2019-07-04

- Adding SOP (Contributed by drkupi)
- Re-enabling restarts: Start a fresh run when we stop making progress

7.6 v.0.2.2, 2019-02-12

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

7.7 v.0.2.1, 2019-01-26

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

7.8 v.0.2.0, 2018-12-06

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

7.9 v.0.1.36, 2017-07-20

- The GUI is now built in PyQt5 instead of PySide

7.10 v.0.1.35, 2017-04-29

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

7.11 v.0.1.34, 2017-03-28

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

7.12 v.0.1.33, 2016-12-27

- Fixed a bug in MARS that resulted in using a lot of zero points for fitting
- Added a GP regression object based on scikit-learn 0.18.1
- Updated tests and documentation

7.13 v.0.1.32, 2016-12-07

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

7.14 v.0.1.31, 2016-11-23

- Added Python 3 support
- Removed Sphinx dependency
- Added six dependency to get py-earth to work for Python 3

7.15 v.0.1.30, 2016-11-18

- Moved all of the official pySOT documentation over to Sphinx
- Five pySOT tutorials were added to the documentation
- The documentation is now hosted on Read the Docs (https://pysot.readthedocs.io)
- Removed pyKriging in order to remove the matplotlib and inspyred dependencies. A new Kriging module will be added in the next version.
- Added the MARS installation to the setup.py since it can now be installed via scikit-learn
- Updated the Sphinx documentation to include all of the source files
- The License, Changes, Contributors, and README files are not in .rst
- Renamed sampling_methods.py to adaptive_sampling.py
- Moved the kernels and tails to separate Python files
- Added a Gitter for pySOT

7.16 v0.1.29, 2016-10-20

- Correcting an error in the pypi upload

7.17 v0.1.28, 2016-10-20

- Making the GUI work with the new RBF design

7.18 v0.1.27, 2016-10-18

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

7.19 v0.1.26, 2016-10-14

- Signficant changes in the RBFInterpolant. Users need to update their code
- Added RBF regression surfaces
- Added version information in the module. pySOT.__version__ gives the version of the current pySOT installation
- The Gutmann strategy has been temporarily removed due to the RBF redesign, but will be added back soon
- Check out test_rbf.py to see how to use the new RBF

7.20 v0.1.25, 2016-09-14

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

7.21 v0.1.24, 2016-08-04

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

7.22 v0.1.23, 2016-07-28

- Updates to support the new MPIController in POAP
- pySOT now sends copies of key variables in case they are changed by the method

7.23 v0.1.22, 2016-06-27

- Added two tests for the MPI controller in POAP
- Removed the accidental matplotlib dependency
- Fixed some printouts in the tests

7.24 v0.1.21, 2016-06-23

- Added an option for supplying weights to the candidate point methods
- Cleaned up some of the tests by appending attributes to the workers
- Extended the MATLAB example to parallel
- Added a help function for doing a progress plot

7.25 v0.1.20, 2016-06-18

- Added some basic input checking (evaluations, dimensionality, etc)
- Added an example with a MATLAB engine in case the optimization problems is in MATLAB
- Fixed a bug in the polynomial regression
- Moved the merit function out of sampling_methods.py

7.26 v0.1.19, 2016-01-30

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

7.27 v0.1.18, 2016-01-24

- Significant restructuring of the code base
- make_points now takes an argument that specifies the number of new points to be generated
- Added Box-Behnken and 2-factorial to the experimental designs
- Simplified the penalty method strategy by moving evals and derivs into a surrogate wrapper

7.28 v0.1.17, 2016-01-13

- Added the possibility to input the penalty for the penalty method in the GUI
- Added the possibility of making a performance plot using matplotlib that adds new points dynamically as evaluations are finished
- Switched from subprocess to subprocess32

7.29 v0.1.16, 2016-01-06

- Added a projection strategy

7.30 v0.1.15, 2015-09-23

- Added an example test_subprocess_files that shows how to use pySOT in case the objective function needs to read the input from a textfile

7.31 v0.1.14, 2015-09-22

- Updated the Tutorial to reflect the changes for the last few months
- Simplified the object creation from strings in the GUI by importing directly from the namespace.

7.32 v0.1.13, 2015-09-03

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

7.33 v0.1.12, 2015-07-23

- The capping can now take in a general transformation that is used to transform the function values. Default is median capping.
- The Genetic Algorithm now defaults to initialize the population using a symmetric latin hypercube design
- DYCORS uses the remaining evaluation budget to change the probabilities after a restart instead of using the total budget

7.34 v0.1.11, 2015-07-22

- Fixed a bug in the capped response surface
- pySOT now internally works on the unit hypercube
- The distance can be passed to the RBF after being computed when generating candidate points so it's not computed twice anymore
- Fixed some bugs in the candidate functions
- GA and Multi-Search gradient perturb the best solution in the case when the best solution is a previously evaluated point
- Added an additional test for the multi-search strategy

7.35 v0.1.10, 2015-07-14

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

7.36 v0.1.9, 2015-07-13

- Fixed a bug in the merit function and several bugs in the DYCORS strategy
- Added a DDS candidate based strategy for searching on the surrogate

7.37 v0.1.8, 2015-07-01

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

7.38 v0.1.7, 2015-06-30

- Fixed some parameters (and bugs) to improve the DYCORS results. Using DYCORS together with the genetic algorithm is recommended.
- Added polynomial regression (not yet in the GUI)
- Changed so that candidate points are generated using truncated normal distribution to avoid projections onto the boundary
- Removed some accidental scikit dependencies in the ensemble surrogate

7.39 v0.1.6, 2015-06-28

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

7.40 v0.1.5, 2015-06-28

- GUI now has support for multiple search strategies and ensemble surrogates
- Reallocation bug in the ensemble surrogates fixed
- Genetic algorithm added to search on the surrogate

7.41 v0.1.4, 2015-06-26

- GUI now has improved error handling
- Strategies informs the user if they get constraints when not expecting constraints (and the other way) before the run starts

7.42 v0.1.3, 2015-06-26

- Experimental (but not documented) GUI added. You need PySide to use it.
- Changes in testproblems.py to allow external objective functions that implement ProcessWorkerThread
- Added GUI test examples in documentation (Ackley.py, Keane.py, SphereExt.py)

7.43 v0.1.2, 2015-06-24

- Changed to using the logging module for all the logging in order to conform to the changes in POAP 0.1.9
- The quiet and stream arguments in the strategies were removed and the tests updated accordingly
- Turned sleeping of in the subprocess test, to avoid platform dependency issues

7.44 v0.1.1, 2015-06-21

- surrogate_optimizer removed, so the user now has to create his own controller
- constraint_method.py is gone, and the constraint handling is handled in specific strategies instead
- There are now two strategies, SyncStrategyNoConstraints and SyncStrategyPenalty
- The search strategies now take a method for providing surrogate predictions rather than keeping a copy of the response surface
- It is now possible for the user to provide additional points to be added to the initial design, in case a 'good starting point' is known.
- Ensemble surrogates have been added to the toolbox
- The strategies takes an additional option 'quiet' so that all of the printing can be avoided if the user wants
- There is also an option 'stream' in case the printing should be redirected somewhere else, for example to a text file. Default is printing to stdout.
- Several examples added to pySOT.test

7.45 v0.1.0, 2015-06-03

- Initial release

CHAPTER 8

License

CHAPTER 9

Contributors

Developed and maintained by:

- David Bindel <bindel@cs.cornell.edu>
- David Eriksson <dme65@cornell.edu>
- Christine Shoemaker <cas12@cornell.edu>

with contributions by:

- Yi Shen <ys623@cornell.edu>
- Taimoor Akhtar <erita@nus.edu.sg>