# SafeOpt Documentation

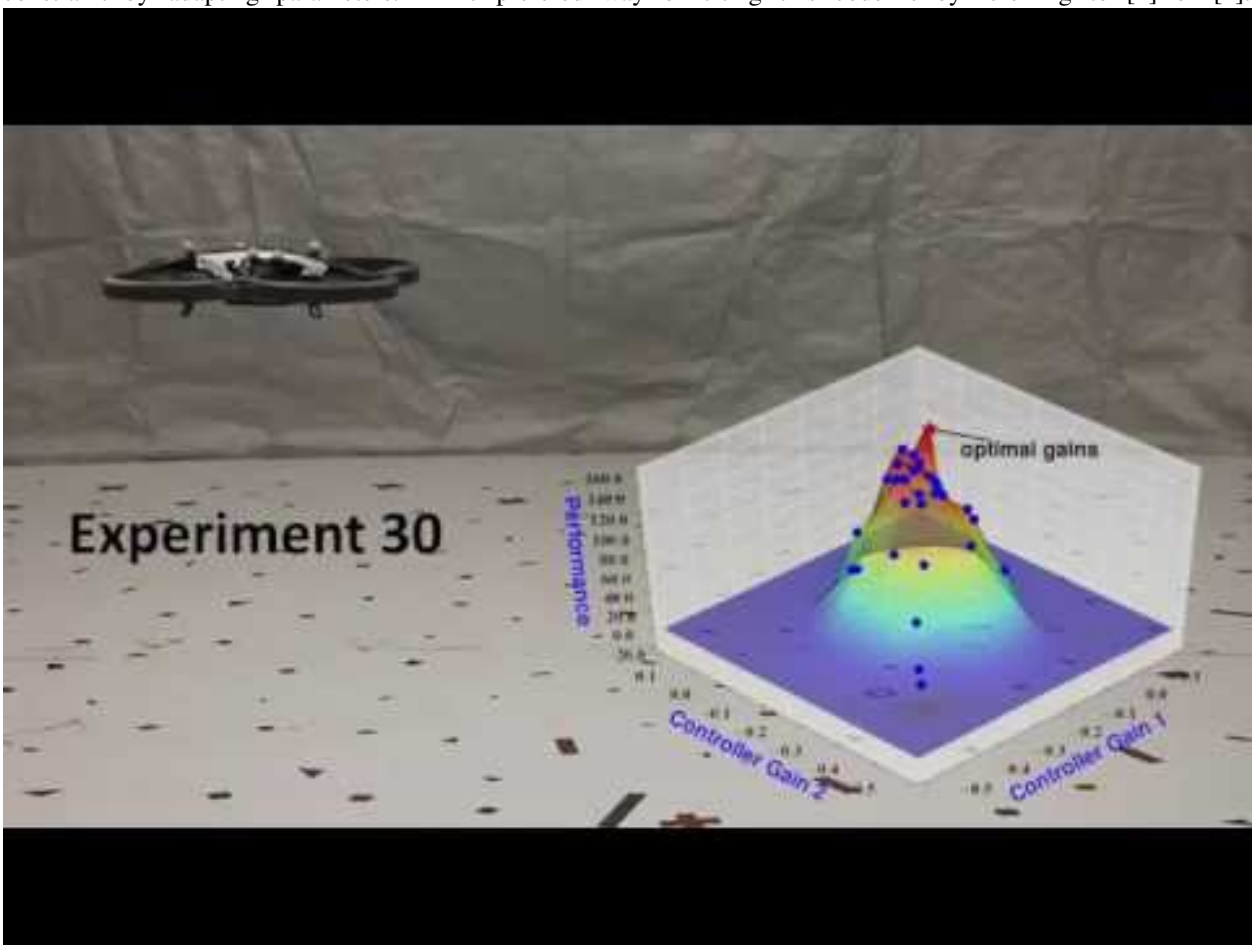
***Release 0.15***

**Felix Berkenkamp, Angela P. Schoellig, Andreas Krause**

**Oct 29, 2018**

# Contents

This code implements an adapted version of the safe, Bayesian optimization algorithm, SafeOpt[1],[2]. It also provides a more scalable implementation based on[3] as well as an implementation for the original algorithm in[4]. The code can be used to automatically optimize a performance measures subject to a safety constraint by adapting parameters. The prefered way of citing this code is by referring to [1] or [2].




[1] F. Berkenkamp, A. P. Schoellig, A. Krause, Safe Controller Optimization for Quadrotors with Gaussian Processes in Proc. of the IEEE International Conference on Robotics and Automation (ICRA), 2016, pp. 491-496.

[2] F. Berkenkamp, A. Krause, A. P. Schoellig, Bayesian Optimization with Safety Constraints: Safe and Automatic Parameter Tuning in Robotics, ArXiv, 2016, arXiv:1602.04450 [cs.RO].

[3] Rikky R.P.R. Duivenvoorden, Felix Berkenkamp, Nicolas Carion, Andreas Krause, Angela P. Schoellig, Constrained Bayesian optimization with Particle Swarms for Safe Adaptive Controller Tuning, in Proc. of the IFAC (International Federation of Automatic Control) World Congress, 2017.

[4] Y. Sui, A. Gotovos, J. W. Burdick, and A. Krause, Safe exploration for optimization with Gaussian processes in Proc. of the International Conference on Machine Learning (ICML), 2015, pp. 997–1005.

CHAPTER 1

# Installation

The easiest way to install the necessary python libraries is by installing pip (e.g. `apt-get install python-pip` on Ubuntu) and running

```
pip install safeopt
```

Alternatively you can clone the repository and install it using

```
python setup.py install
```

CHAPTER 2

# Usage

*The easiest way to get familiar with the library is to run the interactive example ipython notebooks!*

Make sure that the `ipywidgets` module is installed. All functions and classes are documented on Read The Docs.

# CHAPTER 3

# License

The code is licenced under the MIT license and free to use by anyone without any restrictions.

## 3.1 API Documentation

The *safeopt* package implements tools for Safe Bayesian optimization.

### 3.1.1 Main classes

These classes provide the main functionality for Safe Bayesian optimization. `SafeOpt` implements the exact algorithm, which is very inefficient for large problems. `SafeOptSwarm` scales to higher-dimensional problems by relying on heuristics and adaptive swarm discretization.

| | |
|---|---|
| `SafeOpt`(gp, parameter_set, fmin[, ...]) | A class for Safe Bayesian Optimization. |
| `SafeOptSwarm`(gp, fmin, bounds[, beta, ...]) | SafeOpt for larger dimensions using a Swarm Optimization heuristic. |

#### SafeOpt

**class** safeopt.**SafeOpt**(*gp*, *parameter_set*, *fmin*, *lipschitz=None*, *beta=2*, *num_contexts=0*, *threshold=0*, *scaling='auto'*)

A class for Safe Bayesian Optimization.

This class implements the *SafeOpt* algorithm. It uses a Gaussian process model in order to determine parameter combinations that are safe with high probability. Based on these, it aims to both expand the set of safe parameters and to find the optimal parameters within the safe set.

**Parameters**

**gp: GPy Gaussian process** A Gaussian process which is initialized with safe, initial data points. If a list of GPs then the first one is the value, while all the other ones are safety

constraints.

**parameter_set: 2d-array** List of parameters

**fmin: list of floats** Safety threshold for the function value. If multiple safety constraints are used this can also be a list of floats (the first one is always the one for the values, can be set to None if not wanted)

**lipschitz: list of floats** The Lipschitz constant of the system, if None the GP confidence intervals are used directly.

**beta: float or callable** A constant or a function of the time step that scales the confidence interval of the acquisition function.

**threshold: float or list of floats** The algorithm will not try to expand any points that are below this threshold. This makes the algorithm stop expanding points eventually. If a list, this represents the stopping criterion for all the gps. This ignores the scaling factor.

**scaling: list of floats or "auto"** A list used to scale the GP uncertainties to compensate for different input sizes. This should be set to the maximal variance of each kernel. You should probably leave this to "auto" unless your kernel is non-stationary.

### Examples

```
>>> from safeopt import SafeOpt
>>> from safeopt import linearly_spaced_combinations
>>> import GPy
>>> import numpy as np
```

Define a Gaussian process prior over the performance

```
>>> x = np.array([[0.]])
>>> y = np.array([[1.]])
>>> gp = GPy.models.GPRegression(x, y, noise_var=0.01**2)
```

```
>>> bounds = [[-1., 1.]]
>>> parameter_set = linearly_spaced_combinations([[-1., 1.]],
...                                              num_samples=100)
```

Initialize the Bayesian optimization and get new parameters to evaluate

```
>>> opt = SafeOpt(gp, parameter_set, fmin=[0.])
>>> next_parameters = opt.optimize()
```

Add a new data point with the parameters and the performance to the GP. The performance has normally be determined through an external function call.

```
>>> performance = np.array([[1.]])
>>> opt.add_new_data_point(next_parameters, performance)
```

#### Attributes

***context*** Return the current context variables.

***context_fixed_inputs*** Return the fixed inputs for the current context.

***data*** Return the data within the GP models.

***parameter_set*** Discrete parameter samples for Bayesian optimization.

**t** Return the time step (number of measurements).

**use_lipschitz** Boolean that determines whether to use the Lipschitz constant.

**x**

**y**

### Methods

| | |
|---|---|
| add_new_data_point(x, y[, context]) | Add a new function observation to the GPs. |
| compute_safe_set() | Compute only the safe set based on the current confidence bounds. |
| compute_sets([full_sets]) | Compute the safe set of points, based on current confidence bounds. |
| get_maximum([context]) | Return the current estimate for the maximum. |
| get_new_query_point([ucb]) | Compute a new point at which to evaluate the function. |
| optimize([context, ucb]) | Run Safe Bayesian optimization and get the next parameters. |
| plot(n_samples[, axis, figure, plot_3d]) | Plot the current state of the optimization. |
| remove_last_data_point() | Remove the data point that was last added to the GP. |
| update_confidence_intervals([context]) | Recompute the confidence intervals form the GP. |

**add_new_data_point**(*x*, *y*, *context=None*)
Add a new function observation to the GPs.

**Parameters**

**x: 2d-array**

**y: 2d-array**

**context: array_like** The context(s) used for the data points.

**compute_safe_set**()
Compute only the safe set based on the current confidence bounds.

**compute_sets**(*full_sets=False*)
Compute the safe set of points, based on current confidence bounds.

**Parameters**

**context: ndarray** Array that contains the context used to compute the sets

**full_sets: boolean** Whether to compute the full set of expanders or whether to omit computations that are not relevant for running SafeOpt (This option is only useful for plotting purposes)

**context**
Return the current context variables.

**context_fixed_inputs**
Return the fixed inputs for the current context.

**data**
Return the data within the GP models.

**get_maximum**(*context=None*)
Return the current estimate for the maximum.

**Parameters**

**context: ndarray** A vector containing the current context

**Returns**

**x - ndarray** Location of the maximum

**y - 0darray** Maximum value

### Notes

Uses the current context and confidence intervals! Run update_confidence_intervals first if you recently added a new data point.

**get_new_query_point**(*ucb=False*)
Compute a new point at which to evaluate the function.

**Parameters**

**ucb: bool** If True the safe-ucb criteria is used instead.

**Returns**

**x: np.array** The next parameters that should be evaluated.

**optimize**(*context=None*, *ucb=False*)
Run Safe Bayesian optimization and get the next parameters.

**Parameters**

**context: ndarray** A vector containing the current context

**ucb: bool** If True the safe-ucb criteria is used instead.

**Returns**

**x: np.array** The next parameters that should be evaluated.

**parameter_set**
Discrete parameter samples for Bayesian optimization.

**plot**(*n_samples*, *axis=None*, *figure=None*, *plot_3d=False*, ***kwargs*)
Plot the current state of the optimization.

**Parameters**

**n_samples: int** How many samples to use for plotting

**axis: matplotlib axis** The axis on which to draw (does not get cleared first)

**figure: matplotlib figure** Ignored if axis is already defined

**plot_3d: boolean** If set to true shows a 3D plot for 2 dimensional data

**remove_last_data_point**()
Remove the data point that was last added to the GP.

**t**
Return the time step (number of measurements).

**update_confidence_intervals**(*context=None*)
Recompute the confidence intervals form the GP.

**Parameters**

**context: ndarray** Array that contains the context used to compute the sets

**use_lipschitz**
Boolean that determines whether to use the Lipschitz constant.

By default this is set to False, which means the adapted SafeOpt algorithm is used, that uses the GP confidence intervals directly. If set to True, the *self.lipschitz* parameter is used to compute the safe and expanders sets.

### SafeOptSwarm

**class** safeopt.**SafeOptSwarm**(*gp*, *fmin*, *bounds*, *beta=2*, *scaling='auto'*, *threshold=0*, *swarm_size=20*)

SafeOpt for larger dimensions using a Swarm Optimization heuristic.

Note that it doesn't support the use of a Lipschitz constant nor contextual optimization.

You can set your logging level to INFO to get more insights on the optimization process.

**Parameters**

**gp: GPy Gaussian process** A Gaussian process which is initialized with safe, initial data points. If a list of GPs then the first one is the value, while all the other ones are safety constraints.

**fmin: list of floats** Safety threshold for the function value. If multiple safety constraints are used this can also be a list of floats (the first one is always the one for the values, can be set to None if not wanted)

**bounds: pair of floats or list of pairs of floats** If a list is given, then each pair represents the lower/upper bound in each dimension. Otherwise, we assume the same bounds for all dimensions. This is mostly important for plotting or to restrict particles to a certain domain.

**beta: float or callable** A constant or a function of the time step that scales the confidence interval of the acquisition function.

**threshold: float or list of floats** The algorithm will not try to expand any points that are below this threshold. This makes the algorithm stop expanding points eventually. If a list, this represents the stopping criterion for all the gps. This ignores the scaling factor.

**scaling: list of floats or "auto"** A list used to scale the GP uncertainties to compensate for different input sizes. This should be set to the maximal variance of each kernel. You should probably set this to "auto" unless your kernel is non-stationary

**swarm_size: int** The number of particles in each of the optimization swarms

#### Examples

```
>>> from safeopt import SafeOptSwarm
>>> import GPy
>>> import numpy as np
```

Define a Gaussian process prior over the performance

```
>>> x = np.array([[0.]])
>>> y = np.array([[1.]])
>>> gp = GPy.models.GPRegression(x, y, noise_var=0.01**2)
```

Initialize the Bayesian optimization and get new parameters to evaluate

```
>>> opt = SafeOptSwarm(gp, fmin=[0.], bounds=[[-1., 1.]])
>>> next_parameters = opt.optimize()
```

Add a new data point with the parameters and the performance to the GP. The performance has normally be determined through an external function call.

```
>>> performance = np.array([[1.]])
>>> opt.add_new_data_point(next_parameters, performance)
```

**Attributes**

***data*** Return the data within the GP models.

***t*** Return the time step (number of measurements).

**x**

**y**

## Methods

| | |
|---|---|
| *add_new_data_point*(x, y[, context]) | Add a new function observation to the GPs. |
| *get_maximum*() | Return the current estimate for the maximum. |
| *get_new_query_point*(swarm_type) | Compute a new point at which to evaluate the function. |
| *optimize*([ucb]) | Run Safe Bayesian optimization and get the next parameters. |
| *optimize_particle_velocity*() | Optimize the velocities of the particles. |
| *plot*(n_samples[, axis, figure, plot_3d]) | Plot the current state of the optimization. |
| *remove_last_data_point*() | Remove the data point that was last added to the GP. |

**add_new_data_point** (*x*, *y*, *context=None*)
Add a new function observation to the GPs.

**Parameters**

**x: 2d-array**

**y: 2d-array**

**context: array_like** The context(s) used for the data points.

**data**
Return the data within the GP models.

**get_maximum**()
Return the current estimate for the maximum.

**Returns**

**x** [ndarray] Location of the maximum

**y** [0darray] Maximum value

**get_new_query_point** (*swarm_type*)
Compute a new point at which to evaluate the function.

This function relies on a Particle Swarm Optimization (PSO) to find the optimum of the objective function (which depends on the swarm type).

**Parameters**

**swarm_type: string** This parameter controls the type of point that should be found. It can take one of the following values:

- 'expanders' : find a point that increases the safe set
- **'maximizers'** [find a point that maximizes the objective] function within the safe set.
- **'greedy'** [retrieve an estimate of the best currently known] parameters (best lower bound).

**Returns**

**global_best: np.array** The next parameters that should be evaluated.

**max_std_dev: float** The current standard deviation in the point to be evaluated.

**optimize**(*ucb=False*)

Run Safe Bayesian optimization and get the next parameters.

**Parameters**

**ucb: bool** Whether to only compute maximizers (best upper bound).

**Returns**

**x: np.array** The next parameters that should be evaluated.

**optimize_particle_velocity**()

Optimize the velocities of the particles.

Note that this only works well for stationary kernels and constant mean functions. Otherwise the velocity depends on the position!

**Returns**

**velocities: ndarray** The estimated optimal velocities in each direction.

**plot**(*n_samples*, *axis=None*, *figure=None*, *plot_3d=False*, ***kwargs*)

Plot the current state of the optimization.

**Parameters**

**n_samples: int** How many samples to use for plotting

**axis: matplotlib axis** The axis on which to draw (does not get cleared first)

**figure: matplotlib figure** Ignored if axis is already defined

**plot_3d: boolean** If set to true shows a 3D plot for 2 dimensional data

**remove_last_data_point**()

Remove the data point that was last added to the GP.

**t**

Return the time step (number of measurements).

### 3.1.2 Utilities

The following are utilities to make testing and working with the library more pleasant.

| | |
|---|---|
| *sample_gp_function*(kernel, bounds, ... [, ... ]) | Sample a function from a gp with corresponding kernel within its bounds. |
| *linearly_spaced_combinations*(bounds, num_samples) | Return 2-D array with all linearly spaced combinations with the bounds. |
| *plot_2d_gp*(gp, inputs[, predictions, ... ]) | Plot a 2D GP with uncertainty. |
| *plot_3d_gp*(gp, inputs[, predictions, ... ]) | Plot a 3D gp with uncertainty. |
| *plot_contour_gp*(gp, inputs[, predictions, ... ]) | Plot a 3D gp with uncertainty. |

### sample_gp_function

safeopt.**sample_gp_function**(*kernel*, *bounds*, *noise_var*, *num_samples*, *interpolation='kernel'*, *mean_function=None*)

Sample a function from a gp with corresponding kernel within its bounds.

**Parameters**

**kernel: instance of GPy.kern.***

**bounds: list of tuples** [(x1_min, x1_max), (x2_min, x2_max), ... ]

**noise_var: float** Variance of the observation noise of the GP function

**num_samples: int or list** If integer draws the corresponding number of samples in all dimensions and test all possible input combinations. If a list then the list entries correspond to the number of linearly spaced samples of the corresponding input

**interpolation: string** If 'linear' interpolate linearly between samples, if 'kernel' use the corresponding mean RKHS-function of the GP.

**mean_function: callable** Mean of the sample function

**Returns**

**function: object** function(x, noise=True) A function that takes as inputs new locations x to be evaluated and returns the corresponding noisy function values. If noise=False is set the true function values are returned (useful for plotting).

### linearly_spaced_combinations

safeopt.**linearly_spaced_combinations**(*bounds*, *num_samples*)

Return 2-D array with all linearly spaced combinations with the bounds.

**Parameters**

**bounds: sequence of tuples** The bounds for the variables, [(x1_min, x1_max), (x2_min, x2_max), ... ]

**num_samples: integer or array_likem** Number of samples to use for every dimension. Can be a constant if the same number should be used for all, or an array to fine-tune precision. Total number of data points is num_samples ** len(bounds).

**Returns**

**combinations: 2-d array** A 2-d arrray. If d = len(bounds) and l = prod(num_samples) then it is of size l x d, that is, every row contains one combination of inputs.

### plot_2d_gp

safeopt.**plot_2d_gp**(*gp*, *inputs*, *predictions=None*, *figure=None*, *axis=None*, *fixed_inputs=None*, *beta=3*, *fmin=None*, ***kwargs*)

Plot a 2D GP with uncertainty.

**Parameters**

**gp: Instance of GPy.models.GPRegression**

**inputs: 2darray** The input parameters at which the GP is to be evaluated

**predictions: ndarray** Can be used to manually pass the GP predictions, set to None to use the gp directly. Is of the form (mean, variance)

**figure: matplotlib figure** The figure on which to draw (ignored if axis is provided

**axis: matplotlib axis** The axis on which to draw

**fixed_inputs: list** A list containing the the fixed inputs and their corresponding values, e.g., [(0, 3.2), (4, -2.43)]. Set the value to None if it's not fixed, but should not be a plotted axis either

**beta: float** The confidence interval used

**fmin** [float] The safety threshold value.

**Returns**

**axis**

### plot_3d_gp

safeopt.**plot_3d_gp**(*gp*, *inputs*, *predictions=None*, *figure=None*, *axis=None*, *fixed_inputs=None*, *beta=3*, ***kwargs*)

Plot a 3D gp with uncertainty.

**Parameters**

**gp: Instance of GPy.models.GPRegression**

**inputs: 2darray** The input parameters at which the GP is to be evaluated

**predictions: ndarray** Can be used to manually pass the GP predictions, set to None to use the gp directly. Is of the form [mean, variance]

**figure: matplotlib figure** The figure on which to draw (ignored if axis is provided

**axis: matplotlib axis** The axis on which to draw

**fixed_inputs: list** A list containing the the fixed inputs and their corresponding values, e.g., [(0, 3.2), (4, -2.43)]. Set the value to None if it's not fixed, but should not be a plotted axis either

**beta: float** The confidence interval used

**Returns**

**surface: matplotlib trisurf plot**

**data: matplotlib plot for data points**

### plot_contour_gp

safeopt.**plot_contour_gp**(*gp*, *inputs*, *predictions=None*, *figure=None*, *axis=None*, *colorbar=True*, ***kwargs*)

Plot a 3D gp with uncertainty.

**Parameters**

**gp: Instance of GPy.models.GPRegression**

**inputs: list of arrays/floats** The input parameters at which the GP is to be evaluated, here instead of the combinations of inputs the individual inputs that are spread in a grid are given. Only two of the arrays should have more than one value (not fixed).

**predictions: ndarray** Can be used to manually pass the GP predictions, set to None to use the gp directly.

**figure: matplotlib figure** The figure on which to draw (ignored if axis is provided

**axis: matplotlib axis** The axis on which to draw

**Returns**

**contour: matplotlib contour plot**

**colorbar: matplotlib colorbar**

**points: matplotlib plot**

## 3.2 Indices and tables

- genindex
- modindex
- search

# Python Module Index

## S

# Index