PARyOpt Documentation

Release 1.0.2.1

Author

Dec 20, 2021

CONTENTS:

CHAPTER

ONE

INTRODUCTION

We consider a general minimization problem:

$$\min_{\mathbf{x}} \; y(\mathbf{x})$$

Bayesian optimization proceeds through construction of a surrogate cost function $\tilde{y}(\mathbf{x})$. This surrogate is represented as a basis function expansion, around each evaluated point($\mathbf{x}_i, i \in [1, N]$). This ensures that the surrogate passes through (interpolates) the evaluated points. In the case of evaluations with noisy data, the surrogate shall pass within 1 standard deviation from the mean at the evaluated points. Analytically, the surrogate $\tilde{y}(\mathbf{x})$ after N function evaluations is represented as

$$\tilde{y}(\mathbf{x}) = \sum_{i=1}^{N} w_i k(\mathbf{x}, \mathbf{x}_i)$$

where $k(\mathbf{x}, \mathbf{x}_i)$ is a kernel function, i.e., it takes in two arguments, $\mathbf{x}$, $\mathbf{x}_i$ and returns a scalar value. This scalar is representative of how correlated is the function $y(\mathbf{x})$ at $\mathbf{x}$ and $\mathbf{x}_i$. The weights w_i are calculated by solving the system of N linear equations in w_i. In matrix notation, this is represented using a covariance matrix($\mathbf{K}$):

$$\begin{aligned}
\mathbf{K}\,\bar{w} &= y \\
\mathbf{K}_{i,j} &= k(\mathbf{x}_i, \mathbf{x}_j),\ i, j \in [1, N] \\
y_i &= y(\mathbf{x}_i),\ i \in [1, N] \\
\bar{w} &= \{w_i\},\ i \in [1, N]
\end{aligned}$$

Hence the weights are calculated through the inversion $\bar{w} = \mathbf{K}^{-1}\, y$. Note that the covariance matrix $\mathbf{K}$ is a Gram matrix of a positive definite kernel function, making it symmetric and positive semi-definite. Furthermore, since with every iteration only a finite number of rows are added to the covariance matrix, efficient inversion is possible through incremental Cholesky decomposition. The mean and variance of the surrogate are then calculated as:

$$\begin{aligned}
\mu(\mathbf{x}_{N+1}) &= \mathbf{k}^T \mathbf{K}^{-1} y_{1:N} \\
\sigma^2(\mathbf{x}_{N+1}) &= k(\mathbf{x}_{N+1}, \mathbf{x}_{N+1}) - \mathbf{k}^T\, \mathbf{K}^{-1}\, \mathbf{k}
\end{aligned}$$

where

$$\mathbf{k} = k(\mathbf{x}_{1:N}, \mathbf{x}_{N+1}) = [k(\mathbf{x}_1, \mathbf{x}_{N+1})\, k(\mathbf{x}_2, \mathbf{x}_{N+1}) \ldots k(\mathbf{x}_N, \mathbf{x}_{N+1})]$$

At each iteration, the surrogate is updated with new data from the cost function. The locations where the next evaluation is done is determined through optimization of an *acquisition function*. An acquisition function is a means to estimate the new information content at a location. It uses the mean and variance calculated in the above steps.

Some sample radial kernel functions include:

- squared exponential kernel function : Infinitely differentiable

$$k(r) = \theta_0 exp\left(-\frac{r^2}{\theta^2} \right)$$

- Matern class of kernel function :

$$k_{Matern}(r) = \frac{2^{1-\nu}}{\Gamma(\nu)} \left(\frac{\sqrt{2\nu} r}{l} \right)^{\nu} K_{\nu} \left(\frac{\sqrt{2\nu} r}{l} \right)$$

where K_ν is the modified Bessel function, ν, l are positive constants

- Rational quadratic kernel function:

$$k_{RQ}(r) = \left(1 + \frac{r^2}{2\alpha l^2} \right)^{-\alpha}$$

where $r = ||\mathbf{x}_1 - \mathbf{x}_2||$

Some example acquisition functions are:

- Confidence bounds

$$LCB = \mu - \kappa\sigma$$

- Probability of improvement

$$PI = \mathbf{cdf}(\gamma)$$

- Expectation of improvement

$$EI = sqrt(variance) * (\gamma * \mathbf{cdf}(\gamma) + \mathbf{pdf}(\gamma))$$

where $\gamma = \frac{\mu}{\sigma}$, $\mathbf{cdf}$ is cumulative normal distribution function and $\mathbf{pdf}$ is normal probability distribution function

CHAPTER

TWO

INSTALLATION

PARyOpt requires Python 3.5 or above, NumPy, and SciPy for basic functionality. Paramiko is required for cost functions evaluated on remote machines (HPC clusters). Matplotlib is used for visualization for these examples, but is not required.

These all will be installed when you do a

```
pip install paryopt
```

(if you are on Ubuntu, you may need to do `sudo apt-get install python3-pip` and use `pip3` here instead!)

Or, if you prefer an Anaconda environment:

```
conda create -n paryopt python=3.5 numpy scipy matplotlib paramiko
activate paryopt
```

Or, if you are using a manual download:

```
tar -xvf paryopt-1.0.2.1.tar.gz
cd paryopt/
python3.5 setup.py install
```

If you want a virtual environment (preferred), do:

```
tar -xvf paryopt-1.0.2.1.tar.gz
cd paryopt/
python3.5 -m venv venv
source venv/bin/activate
pip install -r requirements.txt
pip install setup.py
```

CHAPTER

THREE

PARYOPT PACKAGE

3.1 Subpackages

3.1.1 PARyOpt.evaluators package

Submodules

PARyOpt.evaluators.paryopt_async module

Asynchronous evaluator super class

class PARyOpt.evaluators.paryopt_async.**AsyncFunctionEvaluator**(*required_fraction: float = 1.0*, *max_pending: int = 0*)

Bases: `object`

Abstract base class for long-running cost functions (e.g. external simulations). Must be subclassed. Subclasses should fill in start() and check_for_results(). Automatically saves state as jobs are submitted.

check_for_result(*x: numpy.core.multiarray.array*, *data: Any*) → Union[*PARyOpt.evaluators.paryopt_async.ValueNotReady*, *PARyOpt.evaluators.paryopt_async.EvaluationFailed*, *PARyOpt.evaluators.paryopt_async.EvaluateAgain*, float]

Returns the cost function evaluation at x, if the value is available. This method is only called after start.

Parameters

- **x** – the point to evaluate the cost function at
- **data** – user data returned by start(x)

Returns an instance of ValueNotReady if such, EvaluationFailed(reason), or the cost function value at x (float)

evaluate_population(*xs: List[numpy.core.multiarray.array]*, *if_ready_xs: List[numpy.core.multiarray.array] = []*) → Tuple[List[Tuple[numpy.core.multiarray.array, float]], List[numpy.core.multiarray.array], List[numpy.core.multiarray.array]]

Evaluates a population of x values, encoded as a list of 1D numpy arrays. Returns a tuple containing three lists:

- Completed values: [(x1, y1), (x2, y2), …]
- Pending values - evaluation is in progress, but not complete: [x1, x2, …]
- Failed values - evaluation completed unsuccessfully: [x1, x2, …]

The union of completed, failed, and pending is equal to the union of xs and if_ready_xs. The __init__ parameter required_fraction tunes how many completed/pending values are returned.

Parameters

- **xs** – list of new points to check
- **if_ready_xs** – List of points to include in the return tuple if they available by the time we evaluate the minimum required percentage of xs. These points do not count towards the minimum required completed points.

Returns ([(x, y), …] completed, [x, …] pending, [x, …] failed)

start(*x: numpy.core.multiarray.array*) → Any

Start a cost function evaluation at the given x location. This method may return anything - the data will be passed on to check_for_result(). The only restriction is that the return value should be pickle-able to enable restart support.

Parameters **x** – point to begin evaluation at

Returns user data that will be fed into check_for_result()

class PARyOpt.evaluators.paryopt_async.**EvaluateAgain**(*reason: str*)

Bases: object

Indicates evaluation needs to be done again, due to some reason (for eg., during hardware failures)

class PARyOpt.evaluators.paryopt_async.**EvaluationFailed**(*reason: str*)

Bases: object

Indicates evaluation was not able to complete successfully, with an error value (i.e. an exception).

class PARyOpt.evaluators.paryopt_async.**ValueNotReady**

Bases: object

Indicates a function value is not ready yet.

PARyOpt.evaluators.async_local module

Local asynchronous evaluator sub-class

class PARyOpt.evaluators.async_local.**AsyncLocalEvaluator**(*job_generator: Callable[[str, numpy.core.multiarray.array], None], run_cmd_generator: Callable[[str, numpy.core.multiarray.array], Union[str, List[Any]]], parse_result: Callable[[str, numpy.core.multiarray.array], float], jobs_dir: str = '/home/balajip/PROJECTS/machineLe required_fraction=1.0, max_pending=8*)

Bases: *PARyOpt.evaluators.paryopt_async.AsyncFunctionEvaluator*

Class for cost functions that evaluated by launching a long-running process on the local machine.

Parameters

- **job_generator** – callable that sets up the run directory for a given x (by e.g. writing config files). It will be passed two arguments: the job directory and the point to evaluate at (x).
- **run_cmd_generator** – callable that returns the command to run the job. It will be passed two arguments: the job directory and the point to evaluate at (x). If run_cmd_generator returns a string, the string will be run by the default shell (typically /bin/sh) via Popen with shell=True. If run_cmd_generator returns a list, it will be passed to Popen. In both cases, the CWD is set to the job directory.
- **parse_result** – callable that returns the cost function evaluated at x. It will be passed two arguments: the job directory and the point to evaluate a t (x). This will be called after the command returned by run_cmd_generator has terminated (gracefully or otherwise). If the process did not terminate successfully or the result is otherwise unavailable, parse_result should raise any exception. This will signal the optimization routine to not try this point again.

- **jobs_dir** – optional base directory to run jobs in - default is $PWD/opt_jobs.
- **required_fraction** – fraction of points which must complete before continuing to the next iteration see AsyncEvaluator for more info and implementation
- **max_pending** – maximum simultaneous processes, defaults to multiprocessing.cpu_count() see AsyncEvaluator for implementation

check_for_result(*x: numpy.core.multiarray.array*, *data: (<class 'str'>, <class 'int'>)*) → Union[*PARyOpt.evaluators.paryopt_async.ValueNotReady*, *PARyOpt.evaluators.paryopt_async.EvaluationFailed*, *PARyOpt.evaluators.paryopt_async.EvaluateAgain*, float]

Checks the status of pid, in data, and calls parse_result if the job is done.

Parameters

- **x** – location of function evaluation
- **data** – list of directory and pid

Returns either ValueNotReady float or EvaluationFailed()

start(*x: numpy.core.multiarray.array) -> (<class 'str'>, <class 'int'>*)

Start a cost function evaluation at the given x location. This method may return anything - the data will be passed on to check_for_result(). The only restriction is that the return value should be pickle-able to enable restart support.

Parameters **x** – point to begin evaluation at

Returns user data that will be fed into check_for_result()

PARyOpt.evaluators.async_parse_result_local module

AsyncLocalParseResultEvaluator An evaluator that only parses files periodically for function evaluations and does not actually execute a script. It periodically checks for the file "job_folder/if_parse.txt" whether the function evaluation is complete or not. If the file contains anything other than 'False', it assumes the external function evaluation is completed and hence checks for "job_folder/y.txt" for the cost function value.

class PARyOpt.evaluators.async_parse_result_local.**AsyncLocalParseResultEvaluator**(*parse_*
Op-
tional[
numpy
float]]
=
None,
job_ge
Op-
tional[
numpy
None]]
=
None,
jobs_d
str
=
'/home
wait_t
date-
time.ti
=
date-
time.ti
60),
to-
tal_fol
int
=
16,
max_p
int
=
0,
re-
quired
float
=
1.0)

Bases: *PARyOpt.evaluators.paryopt_async.AsyncFunctionEvaluator*

Fills files in a set of folders and periodically checks if the external evaluator has finished evaluation. Supports asynchronous evaluations. Only on a local machine.

Variables

- **folder_num** – folder into which the location values are written into
- **job_generator** – callable that sets up the run directory for a given x (by e.g. writing config files). It will be passed two arguments: the job directory and the point to evaluate at (x).
- **jobs_dir** – base directory where the jobs are written into
- **parse_result** – function to parse result from directory , if not specified, it will search in jobs_dir/folder_<folder_num>/if_parse.txt and y.txt
- **total_folders** – total number of folders to write into
- **wait_time** – min time to wait before parsing the results folder

check_for_result(*x: numpy.core.multiarray.array*, *data: (<class 'str'>, <class 'datetime.datetime'>)*) → Union[*PARyOpt.evaluators.paryopt_async.ValueNotReady*, *PARyOpt.evaluators.paryopt_async.EvaluationFailed*, *PARyOpt.evaluators.paryopt_async.EvaluateAgain*, float]

Function to check if a location has been evaluated or not (ValueNotReady). If it completes, categorize the result as one of a float , EvaluationFailed or EvaluateAgain

Parameters

- **x** – location to evaluate
- **data** – directory

Returns Union[ValueNotReady, EvaluationFailed, EvaluateAgain, float]

start(*x: numpy.core.multiarray.array) -> (<class 'str'>, <class 'datetime.datetime'>*)

function to start a cost function evaluation (write to file in this case) given the location of evaluation

Parameters **x** – location of evaluation

Returns folder name in which it was submitted

PARyOpt.evaluators.async_sbatch module

SLURM scheduler asynchronous evaluator sub-class

class PARyOpt.evaluators.async_sbatch.**AsyncSbatchEvaluator**(*host: PARyOpt.evaluators.connection.Host*, *job_generator: Callable[[str, numpy.core.multiarray.array], None]*, *job_script: Union[str, Callable[[str, numpy.core.multiarray.array], str]]*, *lcl_parse_result: Optional[Callable[[str, str, PARyOpt.evaluators.connection.Connecti numpy.core.multiarray.array], float]] = None*, *remote_parse_result: Optional[Callable[[str, numpy.core.multiarray.array], float]] = None*, *lcl_jobs_dir: str = '/home/balajip/PROJECTS/machine squeue_update_rate: datetime.timedelta = datetime.timedelta(0, 30)*, *remote_jobs_dir: str = 'paryopt_jobs'*, *required_fraction=1.0*, *max_pending=25*)

Bases: *PARyOpt.evaluators.paryopt_async.AsyncFunctionEvaluator*

Class for cost functions that evaluated by launching a job on a remote machine running the SLURM job scheduler.

Parameters

- **host** – Host object containing the credentials for the server to connect to
- **job_generator** – callable that sets up the run directory for a given x (by e.g. writing config files). It will be passed two arguments: the job directory and the point to evaluate at (x).

- **job_script** – either a string (for a fixed job script), or a callable that returns the job script string. In the latter case, job_script will be passed two arguments: the job directory and the point to evaluate at (x).
- **remote_parse_result** – callable that returns the cost function evaluated at x. It will be passed two arguments: the job directory and the point to evaluate at (x). This will be called after the command returned by run_cmd_generator has terminated (gracefully or otherwise). If the process did not terminate successfully or the result is otherwise unavailable, parse_result should raise any exception. This will signal the optimization routine to not try this point again. This function will be executed *on the remote host*. This requires the remote host to have a matching version of Python installed and the dill module.
- **lcl_parse_result** – callable that returns the cost function evaluated at X. It is passed three arguments: the local job dir, remote job dir, the Connection object to the remote, and X. It is executed on the local machine. This does not require the remote to have Python installed.
- **lcl_jobs_dir** – optional base directory to generate jobs in - default is $PWD/opt_jobs.
- **remote_jobs_dir** – optional base directory to upload jobs to - default is $HOME/paryopt_jobs.
- **squeue_update_rate** – minimum time between squeue calls. Lower for better job latency, higher to be more polite
- **required_fraction** – fraction of points which must complete before continuing to the next iteration see AsyncEvaluator for more info and implementation
- **max_pending** – maximum simultaneous queued jobs, defaults to 25, see AsyncEvaluator for implementation

check_for_result(*x: numpy.core.multiarray.array*, *data: (<class 'str'>, <class 'str'>, <class 'int'>, <class 'datetime.datetime'>)*) → Union[*PARyOpt.evaluators.paryopt_async.ValueNotReady*, *PARyOpt.evaluators.paryopt_async.EvaluateAgain*, *PARyOpt.evaluators.paryopt_async.EvaluationFailed*, float]

checks for result if the jobid is complete and ping time is after update rate

Parameters

- **x** – location of evaluation
- **data** – data related to the location. Typically this is directory information, job id and submit time

Returns one of ValueNotReady, EvaluateAgain, EvaluationFailed or float

squeue()

start(*x: numpy.core.multiarray.array) -> (<class 'str'>, <class 'str'>, <class 'int'>, <class 'datetime.datetime'>*)

Generate job directory on local machine, fill in data related to the job like directory, job id and submit time

PARyOpt.evaluators.async_sbatch.**VALUE_FROM_FILE**(*filename*)

PARyOpt.evaluators.connection module

class PARyOpt.evaluators.connection.**Connection**

Bases: object

call_on_remote(*remote_func*, **args*, *remote_cwd: Optional[str] = None*)

Call a function created on this system on a remote system with args. This is done by pickling it with dill, SFTPing it to a file on the remote, executing a Python script on the remote that un-dills the file, calls the function, dills the result and prints it to stdout. Finally, stdout is un-dilled on the local machine to give the return value. This requires the remote to have a matching Python version.

Parameters

- **remote_func** – function to call
- **args** – any arguments to call the function with
- **remote_cwd** – directory on the remote to call the script from (must have write access to this directory)

Returns value returned by f

connect(*host:* PARyOpt.evaluators.connection.Host)

exec_command(*cmd: str*, *cwd=None*, *check_exitcode=True*, *encoding='utf-8') -> (<class 'str'>, <class 'str'>, <class 'int'>*)

Executes cmd in a new shell session on the remote host.

Parameters

- **cmd** – command to execute
- **cwd** – directory to execute the command in - performed by prepending 'cd [cwd] && ' to cmd
- **check_exitcode** – if true, instead of returning the exit code of cmd as part of the return tuple, verify that the return code is zero. If it is not, an exception is raised with the contents of stderr.
- **encoding** – encoding to decode stdout/stderr with. Defaults to utf-8.

Returns if check_exitcode is True, (stdout: str, stderr: str). If it is False, (stdout, stderr, rc: int). stdout and stderr are decoded according to encoding.

get_dir(*remote_dir: str*, *local_dir: str*) → None
Download directory from remote directory to local directory

get_file(*remote_path: str*, *local_path: str*) → None
Downloads a file from the remote, same as paramiko.SFTPClient.get

mkdirs(*remote_dir*) → None
Creates remote_dir recursively as a directory on the remote, creating any necessaries parent directories along the way. Similar to *mkdir -p*, except it doesn't error if the directory already exists.

put_dir(*local_path: str*, *remote_path: str*) → None
Compresses local_path into a .tar.gz archive, uploads it to the remote, extracts it into remote_path, and finally deletes the temporary tar archive. Assumes the remote has the 'tar' utility available.

put_file(*local_path: str*, *remote_path: str*) → None
Uploads a local file to the remote host, same as paramiko.SFTPClient.put

remote_python() → str
Returns a string that, when invoked as a command on the remote, will execute a Python that:

- Matches the version that this script was invoked with (i.e. matching sys.version_info)
- Has the 'dill' module installed

The remote Python is discovered by trial and error using common Python names. The search is performed once and then cached. If no such Python is available, this will return None.

sftp() → paramiko.sftp_client.SFTPClient

class PARyOpt.evaluators.connection.**Host**(*username*, *hostname*, *port=22*)
Bases: object

get_interactive(*title: str*, *instructions: str*, *prompts: List[str]*) → List[str]
Handles the ssh 'interactive' authentication mode (user answers a series of prompts).

Parameters

- **title** – title of the window
- **instructions** – instructions, to be shown before any prompts
- **prompts** – the list of prompts

Returns the list of responses (in the same order as the prompts)

get_keys()

Returns a list of public/private keys to try authenticating with

get_password()

Returns the password for the host

Module contents

class PARyOpt.evaluators.**FunctionEvaluator**(*func: Callable[numpy.core.multiarray.array, float]*)

Bases: object

The simplest function evaluator - evaluates a Python function for each point.

Parameters **func** – cost function to evaluate at each x

evaluate_population(*xs: List[numpy.core.multiarray.array]*, *old_xs: List[numpy.core.multiarray.array] = []*) → Tuple[List[Tuple[numpy.core.multiarray.array, float]], List[numpy.core.multiarray.array], List[numpy.core.multiarray.array]]

3.1.2 PARyOpt.kernel package

Submodules

PARyOpt.kernel.kernel_function module

class PARyOpt.kernel.kernel_function.**KernelFunction**

Bases: object

Attributes:

Variables

- **theta** – Kernel parameters. Can be None, a scalar, or a vector. Should be initialized to the appropriate type/len during initialization.
- **theta0** – Scaling factor

derivative(*x1: numpy.core.multiarray.array*, *x2: numpy.core.multiarray.array*) → numpy.core.multiarray.array

Evaluate the derivative of the kernel

Parameters

- **x1** – location 1
- **x2** – location 2

Returns

eval(*x1: numpy.core.multiarray.array*, *x2: numpy.core.multiarray.array*) → float
Evaluate the kernel function

Parameters

- **x1** – location 1
- **x2** – location 2

Returns

theta = None

theta0 = 1.0

PARyOpt.kernel.matern module

class PARyOpt.kernel.matern.**Matern32**
Bases: *PARyOpt.kernel.kernel_function.KernelFunction*

3/2 matern function rbf

derivative(*x1: numpy.core.multiarray.array*, *x2: numpy.core.multiarray.array*) → numpy.core.multiarray.array
Evaluate the derivative of the kernel

Parameters

- **x1** – location 1
- **x2** – location 2

Returns

eval(*x1: numpy.core.multiarray.array*, *x2: numpy.core.multiarray.array*) → float
Evaluate the kernel function

Parameters

- **x1** – location 1
- **x2** – location 2

Returns

theta = 1.0

class PARyOpt.kernel.matern.**Matern52**
Bases: *PARyOpt.kernel.kernel_function.KernelFunction*

5/2 matern function rbf

derivative(*x1: numpy.core.multiarray.array*, *x2: numpy.core.multiarray.array*) → numpy.core.multiarray.array
Evaluate the derivative of the kernel

Parameters

- **x1** – location 1
- **x2** – location 2

Returns

eval(*x1: numpy.core.multiarray.array*, *x2: numpy.core.multiarray.array*) → float
Evaluate the kernel function

Parameters

- **x1** – location 1
- **x2** – location 2

Returns

theta = 1.0

PARyOpt.kernel.squared_exponential module

class PARyOpt.kernel.squared_exponential.**SquaredExponential**
Bases: *PARyOpt.kernel.kernel_function.KernelFunction*

Squared exponential rbf

derivative(*x1: numpy.core.multiarray.array*, *x2: numpy.core.multiarray.array*) → numpy.core.multiarray.array
Evaluate the derivative of the kernel

Parameters

- **x1** – location 1
- **x2** – location 2

Returns

eval(*x1: numpy.core.multiarray.array*, *x2: numpy.core.multiarray.array*) → float
Evaluate the kernel function

Parameters

- **x1** – location 1
- **x2** – location 2

Returns

`theta = 1.0`

Module contents

3.2 Submodules

3.3 PARyOpt.acquisition_functions module

Contains a library of acquisition functions that can be used in bayesian optimization

PARyOpt.acquisition_functions.**expected_improvement**(*mean: float*, *variance: float*, *curr_best: float = 0.0*, *_: float = 1.0*) → float

Expected improvement of objective function 'A Tutorial on Bayesian Optimization of Expensive Cost Functions, with Application to Active User Modeling and Hierarchical Reinforcement Learning'

Parameters

- **mean** – mean of surrogate
- **variance** – variance of surrogate
- **curr_best** – current best evaluated point
- **kappa** – exploration - exploitation tradeoff parameter

Returns expectation of improvement

PARyOpt.acquisition_functions.**lower_confidence_bound**(*mean: float*, *variance: float*, *curr_best: float = 0.0*, *kappa: float = 1.0*) → float

lower confidence bound of improvement : used for minimization problems

Parameters

- **mean** – mean of surrogate
- **variance** – variance of surrogate
- **curr_best** – current best evaluated point
- **kappa** – exploration - exploitation tradeoff parameter

Returns lower confidence bound

PARyOpt.acquisition_functions.**probability_improvement**(*mean: float*, *variance: float*, *curr_best: float = 0.0*, *_: float = 1.0*) → float

Probability of improvement of objective function 'A Tutorial on Bayesian Optimization of Expensive Cost Functions, with Application to Active User Modeling and Hierarchical Reinforcement Learning'

Parameters

- **mean** – mean of surrogate
- **variance** – variance of surrogate
- **curr_best** – current best evaluated point
- **kappa** – exploration - exploitation tradeoff parameter

Returns probability of improvement

PARyOpt.acquisition_functions.**upper_confidence_bound**(*mean: float*, *variance: float*, *curr_best: float = 0.0*, *kappa: float = 1.0*) → float

upper confidence bound of improvement: used in the case of maximization problems

Parameters

- **mean** – mean of surrogate
- **variance** – variance of surrogate
- **curr_best** – current best evaluated point
- **kappa** – exploration - exploitation tradeoff parameter

Returns upper confidence bound

3.4 PARyOpt.paryopt module

Main class for bayesian optimization

class PARyOpt.paryopt.**BayesOpt**(*cost_function: Union[Callable[List[numpy.core.multiarray.array], Tuple[List[Tuple[numpy.core.multiarray.array, float]], List[numpy.core.multiarray.array], List[numpy.core.multiarray.array]]], Callable[[List[numpy.core.multiarray.array], List[numpy.core.multiarray.array]], Tuple[List[Tuple[numpy.core.multiarray.array, float]], List[numpy.core.multiarray.array], List[numpy.core.multiarray.array]]]]*, *l_bound: numpy.core.multiarray.array*, *u_bound: numpy.core.multiarray.array*, *n_dim: int*, *n_opt: int = 1*, *n_init: int = 0*, *init_strategy: Optional[Callable[[int, int, numpy.core.multiarray.array, numpy.core.multiarray.array], List[numpy.core.multiarray.array]]] = None*, *do_init: bool = True*, *kern_function: Optional[Union[str, PARyOpt.kernel.kernel_function.KernelFunction]] = None*, *acq_func: Optional[Union[str, Callable[[float, float, float, float], float]]] = None*, *acq_func_optimizer: Optional[Callable] = None*, *kappa_strategy: Optional[Union[List[Callable[int, float]], Callable[int, float]]] = None*, *constraints: List[Callable[numpy.core.multiarray.array, bool]] = []*, *if_restart: Optional[bool] = None*, *restart_filename: str = 'opt_state.dat'*)

Bases: object

Bayesian optimization class.

Variables

- **curr_iter** – iteration number of optimizer
- **n_surrogate_opt** – number of optima to get during surrogate optimization
- **n_init** – initial population size (>2)
- **n_dim** – dimensions of optimization
- **u_bound** (*l_bound*,) – lower and upper bounds on variables
- **total_population** – total set of population: list of arrays

- **func_vector** – functional values for the population : scalar for each population
- **K** – covariance matrix
- **K_inv** – inverse of covariance matrix
- **K_inv_y** – K_inv * func_vector : pre-computation to save costly / unstable inversion
- **acquisition_function** – acquisition function to optimize the surrogate
- **cost_function** – cost function to MINIMIZE: should be able to take a vector of locations
- **constraints** – list of constraint functions (only points that satisfy these functions will be visited) NOT enforced in the default init strategy!!
- **if_restart** – whether it should restart or not
- **restart_filename** – file to restart from. it will be *opt_state.dat* if nothing is specified
- **acq_func_optimizer** – optimizer for acquisition function. Should take in function, initial guess, bounds and function derivative. Returns the optimal location

acquisition(*x: numpy.core.multiarray.array*, *kappa: float*, *avoid: Optional[List[numpy.core.multiarray.array]] = None*, *opt_indx: int = 0*) → float

Calculates the acquisition function + penalty function at the given query location. If the query point is outside the bounds, it will return an infinity This is the function that needs to be optimized.

Parameters

- **x** – location to evaluate acquisition function : shape: (1, self.n_dim)
- **kappa** – kappa value to pass to acquisition function
- **avoid** – list of x values to avoid (forwarded to penalty function)
- **opt_indx** – index of acquisition function that is being called

Returns scalar acquisition function at x

add_point(*x: numpy.core.multiarray.array*, *y: Optional[float] = None*, *if_check_nearness: bool = True*, *is_failed: bool = False*) → None

Adds ONLY ONE point to the current set of data we have. if_check_nearness specifies if it is needed to check nearness constraints before adding the point into total_population. This should be true if the user is manually adding points into the population. Updates

following variables: K, K_inv, total_population, func_vector If the point is a failed evaluation, then it does not add into total_population and func_vector

Parameters

- **x** – array of shape (1, self.n_dim)
- **y** – cost function value at that location
- **if_check_nearness** – boolean, whether to check nearness or not.
- **is_failed** – boolean, whether the point to add is a failed evaluation or not

Returns None

estimate_best_kernel_parameters(*theta_bounds=None*) → None

Calculates and *sets* the best shape-parameter/characteristic length-scale for RBF kernel function and applies it to the current model.

Parameters **theta_bounds** – array of bounds for theta

Returns None

evaluate_surrogate_at(*x: numpy.core.multiarray.array*, *include_failed: bool = False*) → tuple

Evaluates the surrogate at given point Taken from : https://arxiv.org/pdf/1012.2599.pdf : "A Tutorial on Bayesian Optimization of Expensive Cost Functions, with Application to Active User Modeling and Hierarchical Reinforcement Learning" Note that it returns sigma^2 and not sigma

The mean of the surrogate will not be affected by the failed points, they will only affect the variance of the surrogate

Parameters

- **x** – location to evaluate the surrogate rbf approximation
- **include_failed** – whether to include failed points for surrogate

Returns mean(mu), variance (sigma^2)

export_csv(*path: str*, ***kwargs*) → None

Writes the data from all completed function evaluations (i.e. x, y pairs) to a CSV file. The file starts with the following header row: x0, x1, …, x[n_dim-1], y. Additional arguments will be forwarded to the csv.writer function, which can be used to control the formatting of the CSV file. To write a TSV instead of CSV, you can pass dialect='excel-tab'.

Parameters

- **path** – Path to write the file to. If the file already exists, it will be overwritten.

- **kwargs** – Forwarded to csv.writer(), can be used to override the CSV dialect and set formatting options.

get_current_best() → tuple
returns the location of current best and value of current best

Returns location, func_value

get_current_iteration() → int
Returns the current iteration number

Returns integer

get_total_population() → Tuple[List[numpy.core.multiarray.array], List[float]]
returns the total population : for visualization purposes

Returns total_population and function values at that location

penalty_function(*x: numpy.core.multiarray.array*, *kappa: float*, *avoid: List[numpy.core.multiarray.array] = []*) → float
Calculates the penalty function, to prevent local optima to repeat

Parameters

- **x** – location
- **kappa** – kappa value used in acquisition function
- **avoid** – extra points to avoid (other than self.total_pop/self.pending/self.failed)

Returns penalty function value at x

set_new_kernel_parameters(*theta: Union[float, numpy.core.multiarray.array] = 1.0*, *theta0: float = 1.0*) → None
sets the kernel parameters: called from the user script. The covariance matrix needs to be re-calculated when this function is called

Parameters

- **theta** – scaling of dimensions used with the distance function
- **theta0** – scaling of kernel function

Returns None

update_iter(*iter_max: int = 1*) → None
Finds the next query point and updates the data for the next iteration

Returns None

update_surrogate() → None
updates the inverse of covariance function. In order to incorporate failed locations as interpolated values, this update is 2 staged:

1. Create the surrogate (K,K_inv) with only the successfully evaluated points
2. a) Evaluate the mean of the surrogate at the failed points

   b) Incorporate the interpolated failed locations into the surrogate and update the surrogate (K,K_inv)

Updates the values of self.K_success, self.K_inv_success, self.K_inv_y_success, self.K_inv and self.K_inv_y

Returns None

3.5 PARyOpt.utils module

Utility functions that will be used in kernels, acquisition functions and bayes optimization

PARyOpt.utils.**cdf_normal**(*x: numpy.core.multiarray.array*, *mean: numpy.core.multiarray.array = 0.0*, *sigma_sq: numpy.core.multiarray.array = 1.0*) → float

Cumulative distribution function of standard normal distribution

Parameters

- **x** – scalar / location
- **mean** – mean of distribution
- **sigma_sq** – variance of distribution (sigma^2)

Returns cdf of normal distribution

PARyOpt.utils.**distance**(*x1: numpy.core.multiarray.array*, *x2: numpy.core.multiarray.array*) → float

returns the distance between two query points

Parameters

- **x1** – point 1
- **x2** – point 2

Returns euclidean distance between the two points

PARyOpt.utils.**erf**(*x*)

error function of x: used in calculating cumulative distribution. Unable to import from scipy, so writing our own function

Parameters **x** – float

Returns error function of x

PARyOpt.utils.**lhs**(*n: int*, *samples: Optional[int] = None*, *criterion: Optional[str] = None*, *iterations: Optional[int] = None*)

Generate a latin-hypercube design

Parameters

- **n** – The number of factors to generate samples for
- **samples** – The number of samples to generate for each factor (Default: n)
- **criterion** – Allowable values are "center" or "c", "maximin" or "m", "centermaximin" or "cm", and "correlation" or "corr". If no value given, the design is centermaximin.
- **iterations** – The number of iterations in the maximin and correlations algorithms (Default: 5).

Return H An n-by-samples design matrix that has been normalized so factor values are uniformly spaced between zero and one.

Example

A 3-factor design (defaults to 3 samples):

```
>>> lhs(3)
array([[ 0.40069325,  0.08118402,  0.69763298],
       [ 0.19524568,  0.41383587,  0.29947106],
       [ 0.85341601,  0.75460699,  0.360024  ]])
```

A 4-factor design with 6 samples:

```
>>> lhs(4, samples=6)
array([[ 0.27226812,  0.02811327,  0.62792445,  0.91988196],
       [ 0.76945538,  0.43501682,  0.01107457,  0.09583358],
       [ 0.45702981,  0.76073773,  0.90245401,  0.18773015],
       [ 0.99342115,  0.85814198,  0.16996665,  0.65069309],
       [ 0.63092013,  0.22148567,  0.33616859,  0.36332478],
       [ 0.05276917,  0.5819198 ,  0.67194243,  0.78703262]])
```

A 2-factor design with 5 centered samples:

```
>>> lhs(2, samples=5, criterion='center')
array([[ 0.3,  0.5],
       [ 0.7,  0.9],
       [ 0.1,  0.3],
       [ 0.9,  0.1],
       [ 0.5,  0.7]])
```

A 3-factor design with 4 samples where the minimum distance between all samples has been maximized:

```
>>> lhs(3, samples=4, criterion='maximin')
array([[ 0.02642564,  0.55576963,  0.50261649],
       [ 0.51606589,  0.88933259,  0.34040838],
       [ 0.98431735,  0.0380364 ,  0.01621717],
       [ 0.40414671,  0.33339132,  0.84845707]])
```

A 4-factor design with 5 samples where the samples are as uncorrelated as possible (within 10 iterations):

```
>>> lhs(4, samples=5, criterion='correlate', iterations=10)
```

PARyOpt.utils.**pdf_normal**(*x: numpy.core.multiarray.array*, *mean: numpy.core.multiarray.array = 0.0*, *sigma_sq: float = 1.0*) → float

Probability distribution function of standard normal distribution, returns the pdf of a location from given mean with variance sigma_sq

Parameters

- **x** – scalar/location
- **mean** – mean of distribution
- **sigma_sq** – variance of distribution (sigma^2)

Returns pdf of normal distribution

3.6 Module contents

CHAPTER

FOUR

DOCS

4.1 conf module

class conf.**Mock**(**args*, ***kw*)

Bases: unittest.mock.MagicMock

CHAPTER

FIVE

EXAMPLES

This provides a demonstration of the exhaustive functionality of PARyOpt. These are very intricately connected to the examples on BitBucket, so the user is suggested to go through them simultaneously. The examples are structured as follows:

Table 1: Description of Examples

Example #	Description
Example **0** *Getting Started*	A dive into setting up the optimization routine.
Example **1** *Custom functions for surrogate construction*	Using the same problem as example 0, the modularity of the framework is demonstrated. All the functions that can be customized are shown.
Example **2** *Restart from previous state*	Explains the restart capabilities of the framework and how one can use in case of (hardware/resource) failure.
Example **3** *Local asynchronous evaluator*	Asynchronocity is introduced and a local asynchronous implementation of example 0 is shown. This implementation can be easily extended to an asynchronous remote evaluator.
Example **4** *Kriging*	Explains the kriging functionality of the framework and how data can be assimilated to perform kriging

5.1 Getting Started

This example shows how one gets started with the optimization software. One is expected to see through **example_0.py** for a better understanding.

Just like any optimization, we should have information about the following:

1. dimensionality of the problem – `n_dim`
2. cost function that has to be **minimized** – `function`
3. bounds on the variables – `l_bound`, `u_bound`

Some more parameters for Bayesian Optimization

4. number of initial evaluations for constructing the prior –`n_init`
5. type of kernel function – `kern_function`
6. number of evaluations per iteration – `n_opt`
7. platform of evaluation - local computer / remote computer
8. parallel / serial evaluations – `cost_function`
9. asynchronocity of evaluations – `cost_function`
10. acquisition function (list) – `acq_func`
    - parallelization of acquisition function – `kappa_strategy`

The `kappa_strategy` defines exploration vs exploitation of the optimizer. Those with a *large* `kappa` value will explore and a *small* `kappa` value will exploit.

In this example, we shall solve a simple parabolic cost function, on a local machine with no parallelization. The evaluations will, therefore, be fully synchronous. As part of the example, we shall do an exploration dominated search in the optimization. The parameters that will be used are as follows:

```
from PARyOpt.evaluators import FunctionEvaluator
import numpy as np
from PARyOpt import BayesOpt

n_dim = 1
l_bound = np.asarray([-12.])
u_bound = np.asarray([12.])

n_init = 2
kern_function = 'sqr_exp'       # squared exponential
acq_func = 'LCB'                # lower confidence bound
def my_cost_function(x: np.array) -> float:
        y = np.sum((x-2.5) ** 2 + 5)
        return float(y)
# instantiate an evaluator that evaluates serially on the local machine
evaluator = FunctionEvaluator(my_cost_function)
def my_kappa(curr_iter: int) -> float:
    return 1000.0           # large value for exploration
```

5.1.1 Initialization

Having defined these parameters, we shall now initialize the optimizer:

```
b_opt = BayesOpt(cost_function=evaluator,
                 l_bound=l_bound, u_bound=u_bound, n_dim=n_dim,
                 n_init=2,
                 kern_function='sqr_exp',
                 acq_func='LCB',
                 n_opt=1,                              # default setting
                 kappa_strategy=my_kappa,
                 if_restart=False)
```

The stage is now set for optimization to be performed. Since this package does not provide any standard termination criteria, the user is expected to design a termination based on the nature of the problem. In this example, we shall look at a very simple termination criterion of number of iterations.

```
max_iter = 10
```

5.1.2 Update

The user shall manually update the optimization every iteration. This provides ways to post-process user required metrics every iteration, as well as do a regular *hyper-parameter optimization* for optimized surrogate.

```
for curr_iter in range(max_iter):
    b_opt.update_iter()
```

5.1.3 Hyper parameter optimization

An implementation of the standard hyper parameter optimization is done in `estimate_best_kernel_parameters()`. This minimizes a maximum likelihood estimate of the constructed surrogate and eventually sets the *optimal* kernel length scale. It can be invoked by calling:

```
theta_min = 0.01
theta_max = 50.
b_opt.estimate_best_kernel_parameters(theta_bounds=[[theta_min, theta_
↪max]])
```

Hyper parameter optimization need not be performed every iteration as the surrogate may not change much with the addition of a single data point. Hence its call can be periodic based on the iteration number.

5.1.4 Surrogate query

Having constructed the surrogate, one may need to query it for several purposes, including visualization, post-processing and termination criteria. This functionality is provided through the `evaluate_surrogate_at()` function. It returns the value of the mean and variance of the surrogate at the queried location.

```
location_to_query = np.asarray([0.5])
mean, variance = b_opt.evaluate_surrogate_at(location_to_query)
```

5.1.5 Logging

PARyOpt uses the python logging module for logging. The user has to instantiate the logger in the main code. If the logger is not initiated, the logs will be streamed to `stdout`. An example of using the logger is also in the above example:

```
import logging, time

logger = logging.getLogger()
logger.setLevel(logging.INFO)  # either NOTSET, INFO, DEBUG, WARNING,␣
↪ERROR, CRITICAL -- different levels of log
log_file_name = 'example0_{}.log'.format(time.strftime("%Y.%m.%d-%H%M%S"))
fh = logging.FileHandler(log_file_name, mode='a')
# logging format
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s -
↪%(message)s')
fh.setFormatter(formatter)
logger.addHandler(fh)
```

5.1.6 Saving data

The framework provides multiple ways to save data, particularly with ready methods to export in `.csv` format. It can be done be calling:

```
b_opt.export_csv('my_data.csv')
```

Alternately, data can be custom exported, as `get` methods exist to get the population and the respective function values.

```
total_population, function_values = b_opt.get_total_population()
```

The next example shows how to custom change the various functions used in the optimization method.

5.2 Custom functions for surrogate construction

This example discusses the modularity provided by the framework to change various optimizer and surrogate related methods. More specifically, we look at changing the following accessory functions:

- *Initialization strategy*
- *Kernel functions (Class)*
- *Acquisition functions*
- *Acquisition optimizer*

Through this tutorial, we shall not solve any new problem. However, explanations will be provided to how can one add custom definitions to these standard methods. We shall use an external implementation of the standard methods for the user to easily compare with *Getting Started*. While the example shows the usage of all of these at once, the user is encouraged to understand the effect of changing each of these accessory functions separately:

5.2.1 Initialization strategy

Several situations arise when the user has to specify his own initialization strategies. Some of these include:

- **Constraints**, such as equality, inequality and PDE based constraints. In such situations, all locations determined by the upper and lower bounds may not be feasible. PARyOpt requires all the initial evaluation points to be successful evaluations, hence a general latin hypercube sampling may not work in all cases.
- **Biased sampling** of search domain. There could be situations where the user knows strategic locations and thus can help PARyOpt to sample in those locations right from the beginning
- **User defined sampling strategy**. The user may want a completely different sampling strategy which has a better coverage of the domain for that specific optimization problem.

In order to specify custom initialization strategy, the user should define a function with the following signature:

```
def my_init_strategy(n_dim: int, n_init: int, l_bound: np.array, u_bound:␣
↪np.array) -> List[np.array]
```

and then pass it during initialization to the parameter `init_strategy=my_init_strategy`.

5.2.2 Kernel functions (Class)

Kernel function embeds most of the information about the **continuity** of the underlying function. Thus, it is one of the critical parameters to be selected by the user. For example, Matern class of kernel functions have only finite differentiability while the squared exponential kernel function is infinitely differentiable. Some other situations where the user may want to change the kernel function are :

- **Underlying function continuity and differentiability**, as discussed above.
- **Prior information about periodicity** of the underlying function. In such situations, periodicity can be embedded into the kernel function. This can drastically improve the number of function evaluations. One such example of a periodic kernel function is:

$$k_{periodic}(x_1, x_2) = \theta_0 exp(-2sin(\frac{\pi}{p}\frac{||x_1 - x_2||}{l})^2)$$

where p is the periodicity interval and l is the length scale of the kernel.

- **Anisotropic kernels** may be used when the function behaviour is different across different optimization parameters. For example, if the function varies logarithmically in one parameter and linearly in another parameter

To specify custom kernel functions, the user has to derive the base kernel class. An example to specify the standard Matern 3/2 function by the user is as follows:

```
class MyKernelFunction(KernelFunction):
 """
 user customized kernel function
 """

 theta = 1.0

 def eval(self, x1: np.array, x2: np.array) -> float:
     """
     actual kernel function
     :param x1:
     :param x2:
     :return:
     """
     x1 = np.divide(x1, self.theta)
     x2 = np.divide(x2, self.theta)
     dist = utils.distance(x1, x2)
     rval = np.sqrt(3.0) * dist
```

(continues on next page)

(continued from previous page)

```
    return self.theta0 * (1+rval) * np.exp(-rval)

def derivative(self, x1: np.array, x2: np.array) -> np.array:
    """
    derivative of kernel function
    currently not useful, so we will not implement anything here
    :param x1:
    :param x2:
    :return:
    """
    pass
```

This derived class requires defining the `eval()` and `derivative()` methods of the class. The parameter `theta` should contain all the hyper-parameters, for ex. length scale, related to the supplied kernel. This will be used by PARyOpt during hyper-parameter optimization. This kernel class can be passed to the constructor through the parameter `kern_function=MyKernelFunction()`.

5.2.3 Acquisition functions

Acquisition function define informative regions of the surrogate. Since the software is designed for minimization, areas of high information should have small acquisition values. In normal circumstances, the user should not need to change this. However, if the user wants implementing **safety constraints** and **biased informativeness**, it can be be done by passing a user defined function, with the following signature, to the constructor as `acq_func=my_acquisition_function`.

```
def my_acquisition_function(mean: float, variance: float, curr_best:␣
↪float = 0., kappa: float = 1.) -> float:
```

5.2.4 Acquisition optimizer

Bayesian optimization proceeds by evaluating locations with maximum information content. Hence, it requires finding the optimum (minimum in PARyOpt) of the acquisition function (cheap to evaluate). While the core software comes with a standard `Powell` algorithm, the user may quite often want to change this for a better global optimum. Some of the reasons include:

- **Dimensionality of optimization** could impose restrictions on the type of optimizer used
- **Heuristic optimization** can be an alternative for multi-modal functions. Since evaluating the acquisition is fast, these optimizers will also be efficient.
- Robust **in-house optimizers** may be available with research groups tailored for specific problems.

The process of adding user-defined acquisition optimizer is very similar to defining custom acquisition function. The function signature is :

```
def my_acq_optimizer(func: Callable[[np.array], float], x0: np.array, l_
↪bound: np.array, u_bound: np.array) -> np.array:
```

which is passed to the constructor as `acq_func_optimizer=my_acq_optimizer`.

5.3 Restart from previous state

This example shows how the user can deal with optimization failures, either due to hardware failure or due to wrong selection of accessory functions. This will also be useful in cases of changing optimization platform but resuming the same optimization task.

In order to create an optimization state, we shall first run a standard optimization problem for a certain number of iterations. This can be done either by running example 0(*Getting Started*), or by re-doing the whole procedure. For the benefit of the user, we shall take the latter way.

5.3.1 Restart

In example 1, we have seen that our custom initialization is not the best compared to the default latin hypercube sampling. In fact, such an example provides the best motivation for a restart. Hence, in this example, we provide a custom initialization method, the same as in example 1 (*Custom functions for surrogate construction*). Once the optimization is done for 10 iterations, we shall create another instance of `BayesOpt` that starts from this existing optimization state.

```
restarted_bo = BayesOpt(cost_function=evaluator,
                        l_bound=l_bound, u_bound=u_bound, n_dim=n_dim,
                        n_init=2,
                        kern_function='sqr_exp',
                        acq_func='LCB',
                        kappa_strategy=my_kappa,
                        if_restart=True, restart_filename='opt_state.dat')
```

Note that the restarted optimization need not have the same accessory functions, like kernel, acquisition and kappa strategy. By enabling `if_restart` and providing the `restart_filename`, the framework re-creates the optimization state from which the user can continue the optimization, for example,

```
restarted_bo.update_iter(5)
```

will update 5 iterations at once. This API helps to reduce redundant loops in the user code.

5.3.2 Intra-iteration restart

It has to be noted here that the evaluator also has an inbuilt check-pointing per iteration, so that hard interrupts such as the `KeyboardInterrupt` can also be handled for restart. The user need not do any extra changes to enable this **intra-iteration** restart functionality.

5.4 Local asynchronous evaluator

In this example, we shall see how to use the local async evaluator. This is particularly useful when the cost function takes a long time to evaluate, and sometimes with uncertain evaluation times. In such situations, a better way to parallelize evaluations is to run as individual jobs on the evaluation platform (local machine) and keep track of the completion of the jobs through the process id in the process table. This approach naturally supports asynchronous bayesian update, i.e., update the iteration without completing all the jobs per iteration. The un-assimilated jobs will be used for the update in the subsequent iterations.

To achieve this, the framework creates individual job directories for each cost function evaluation. It also provides an interface for the user to set up these directories for a *self-contained* evaluation. The cost function needs to write out a file with the result(cost value) which will later be parsed by the framework upon execution completion. For this, the user needs to provide functions that

1. generate necessary files in the folder for function evaluation
2. provide the command for executing the cost function, and
3. parse the result file generated by the cost function

In the rest of the document, we shall see how to set up a sample cost function, a folder generator, run command and result parser. Once again, we shall use the same parabolic cost function for easy understanding, but in 2 dimensions.

5.4.1 Out-of-script cost function

The first step to performing out-of-script evaluation is to re-define the cost function. While a generic cost function in an optimization framework is like a python function that returns the cost function value, out-of-script functions need to be defined differently. Firstly, they do not have any defined arguments and secondly, such out-of-script cost function also cannot directly pass the function value to the optimization framework. While there are several ways to overcome these difficulties, this framework requires the following template to be followed:

- Input: the cost function can either take inline arguments or read from file
- Output: the cost function should write to a standard file

For example, the following code evaluates the parabola in 2 dimensions and writes to file `result.txt`. Location of evaluation are passed as inline arguments.

```
import sys
import numpy as np
import time
import examples.examples_all_functions as exf

# read the command line arguments into an array
xs = np.asarray([float(x) for x in sys.argv[1:]])

# evaluate the cost, i.e. the parabola
cost = exf.parabolic_cost_function(x=xs)

# In order to demonstrate uncertain evaluation times, we shall use a␣
↪random sleep in each cost function.
# In this case, each function evaluation can take between 0-10 sec
time.sleep(np.random.random() * 10.)

# Write into a result file. Note that this script is evaluated in its␣
↪respective folder and so
# the result.txt file will be in the generated folder and not the home␣
↪directory of running example4.py
with open('result.txt', 'w') as f:
    f.write(str(cost) + '\n')
```

5.4.2 Folder generator

Many common simulations require not just the location of evaluation but also several other systems to be in place for proper working. For example, many finite element simulations require a geometry mesh file that represent the domain of simulation. The optimizer calls this function, `job_generator()` with two arguments – the folder of evaluation (more to come on this) and the location `x` at which the cost function is executed.

```
def folder_generator(directory, x) -> None:
    """
    prepares a given folder for performing the simulations. The cost␣
↪function (out-of-script) will be executed
    in this directory for location x. Typically this involves writing a␣
↪config file, generating/copying meshes and

    In our example, we are running a simple case and so does not require␣
↪any files to be filled. We shall pass the
    location of cost function as a command line argument
    """
```

(continues on next page)

(continued from previous page)

```
    with open(os.path.join(directory, 'config.txt'), 'w') as f:
        pass  # write file
    pass
```

5.4.3 Run command

Just like a folder of files for execution, the user may need to provide command line arguments to the cost function during execution. To achieve this, the optimizer calls the function `run_cmd_generator()` with the folder of evaluation and location `x` at which the cost function has to be evaluated. Thus it can allow change of run-time arguments based on the evaluation point.

```
def run_cmd(directory, x) -> List[Any]:
    """
    Command to run on local machine to get the value of cost function at␣
↪x, in directory.
    In this example, we shall run the script example3_evaluator.py with␣
↪the location as an argument.
    """
    eval_path = os.path.join(os.path.dirname(os.path.abspath(__file__)),
↪'example3_evaluator.py')
    return [sys.executable, eval_path] + list(x[:])
```

5.4.4 Result parser

Once the cost function writes the cost value into a file, the result parser is supposed to read and return that cost value to the optimizer. Some local post processing operations can go into this function. Care should be taken to return **only float values**, otherwise it can lead to Type inconsistencies in the optimization routine. The function signature is the same as that for `run_cmd_generator()` and `job_generator()`

```
def result_parser(directory, x) -> float:
    """
    Parses the result from a file and returns the cost function.
    The file is written be the actual cost function. One can also do post␣
↪processing in this function and return the
    subsequent value. Based on the construct of our cost function␣
↪example3_evaluator.py, the generated result.txt
    will be in this 'directory'
    """
    with open(os.path.join(directory, 'result.txt'), 'r') as f:
        return float(f.readline())
```

5.4.5 Asynchronous optimization

Once the above functions are created, the only new procedure to use asynchronous evaluations is *setting up the evaluator*. This requires passing in the the three functions, namely, `job_generator()`, `run_cmd_generator()` and `parse_result()`. Along with these, it is optional to pass in the location of function evaluations. The `evaluator` creates separate folders in these directory (relative path) for each cost function evaluation. Each cost function call is assigned a (randomly named) directory within this specified `jobs_dir` where the `run_cmd` (from `run_cmd_generator()`) is called.

```
evaluator = AsyncLocalEvaluator(job_generator=folder_generator,
                                run_cmd_generator=run_cmd,
                                parse_result=result_parser,
                                required_fraction=0.5, jobs_dir=os.path.
↪join(os.getcwd(), 'temp/opt_jobs'))
```

Since we are using multiple optima per iteration, we can take advantage of it deploy simultaneous exploration and exploitation in the acquisition function. For example, the following code creates a list of two functions – one exploratory (kappa = 1000) and another exploitatory (kappa = 0.1). This list is then passed to the optimizer, like in the previous examples.

```
n_opt = 2
my_kappa_funcs = []
my_kappa_funcs.append(lambda iter_num: 1000)            # exploration
my_kappa_funcs.append(lambda iter_num: 0.1)             # exploitation
```

One can get more crafty in designing these kappa strategies and create a so-called annealing kappa, one that starts with a large value and eventually reduces to a small value, at different rates.

```
for j in range(n_opt):
    my_kappa_funcs.append(lambda curr_iter_num, freq=10.*(j*j+2), t_
↪const=0.8/(1. + j):
                          user_defined_kappa(curr_iter_num, freq=freq, t_
↪const=t_const))
```

The remaining part of the optimization remains the same, except for the initialization of `BayesOpt` object.

```
b_opt = BayesOpt(cost_function=evaluator,
                 n_dim=n_dim, n_opt=n_opt, n_init=2,
                 u_bound=u_bound, l_bound=l_bound,
                 kern_function='matern_52',
                 acq_func='LCB', kappa_strategy=my_kappa_funcs,
                 if_restart=False)
```

(continues on next page)

(continued from previous page)

```
for curr_iter in range(iter_max):
    b_opt.update_iter()
    if not curr_iter % 2:
        b_opt.estimate_best_kernel_parameters(theta_bounds=[[0.01, 10]])
    exf.visualize_fit(b_opt)
```

5.5 Kriging

Kriging or Gaussian process regression is a method of interpolation for which the interpolated values are modeled by a Gaussian process governed by prior covariances. In this example, we show how PARyOpt can be used to generate response surfaces using available data. As with the previous examples, we shall use the standard parabola as the underlying function to be approximated. There are several ways to use PARyOpt for Kriging, one of which is shown here. This is possibly the easiest and cleanest way to perform Kriging using PARyOpt.

Data generation: Since the underlying function is known, we shall generate data by invoking this function at some random locations within the bounds and storing them in an external file. This is achieved through the following snippet:

```
def create_data_csv(function: Callable, filename: str, l_bound: np.array,␣
↪u_bound: np.array) -> None:
 # generate some random locations -- 7
 normalized_population = np.random.ranf((7, ))
 real_population = l_bound + normalized_population * (u_bound - l_bound)
 real_population = [np.asarray([p]) for p in real_population]
 # evaluate the values
 real_functions = [float(function(p)) for p in real_population]

 # write into file
 with open(filename, 'w') as f:
     writer = csv.writer(f, delimiter=',')
     writer.writerow('x y')
     for x, y in zip(real_population, real_functions):
         writer.writerow(list(x) + [y])
```

Data assimilation : PARyOpt provides an `add_point()` to add external data manually. The user has to supply the x location and any available y values to this method to add data to the `BayesOpt` instance. An example usage of `add_point()` using the above generated data can be:

```
def load_from_csv(b_opt: BayesOpt, filename: str) -> BayesOpt:
 """
```

(continues on next page)

(continued from previous page)

```
    load data from csv file and add to PARyOpt
    """
    with open(filename, 'r') as csvfile:
        csv_file_lines = csv.reader(csvfile, delimiter=',')
        for row_num, row in enumerate(csv_file_lines):
            if row_num == 0:
                # skipping the header
                pass
            else:
                b_opt.add_point(x=np.asarray([float(row[0])]), y=float(row[-
↪1]),
                                if_check_nearness=True)
    b_opt.update_surrogate()

    return b_opt
```

Note that the user has to manually invoke `update_surrogate()`. This is currently for efficiency purposes and hope to be replaced in the upcoming versions.

Finally, since the user wants to add data manually and does not want the standard initialization required for bayesian optimization, we provide a switch `do_init` to turn off the initialization. Since there is no cost function to be optimized, the evaluator should be passed in an empty function for evaluation.

```
# dummy evaluators:
evaluator = FunctionEvaluator(lambda x: 0.0)

krig = BayesOpt(cost_function=evaluator,
                l_bound=l_bound, u_bound=u_bound, n_dim=1,
                n_init=0, do_init=False,        # ensures that␣
↪initialization is not done.
                kern_function='sqr_exp',
                acq_func='LCB',
                kappa_strategy=lambda curr_iter: 1000,
                if_restart=False)
krig = load_from_csv(krig, data_filename)
krig.estimate_best_kernel_parameters(theta_bounds=[[0.001, 10.0]])
```

Note that since we are not providing any actual cost function here, `update_iter()` does nothing useful. In case the user is looking for an instantaneous Kriging model, i.e., creating a Kriging surface and updating it, the actual cost function should be provided. Just like the previous examples, one may use `evaluator` from *Local asynchronous evaluator* and do Kriging similar to optimization.

Now that the surrogate is created and hyper-parameters optimized, one can start querying it using

`evaluate_surrogate_at()`

```
location = np.array([1.0])
mean, variance = krig.evaluate_surrogate_at(location)
```

PARyOpt (pronounced **pur-yopt**, with a hard *y*) is a modular asynchronous Bayesian optimization package that enables remote function evaluation.

It is written to solve optimization problems where the cost function is very expensive to evaluate (on the order of hours), and where cost functions must be evaluated on remote machines (HPC clusters).

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

c

p

INDEX

P

R

S

T

U

V