M-LOOP Documentation

Release 2.1.0

Michael R Hush

Jun 25, 2020

Contents

The Machine-Learner Online Optimization Package is designed to automatically and rapidly optimize the parameters of a scientific experiment or computer controller system.

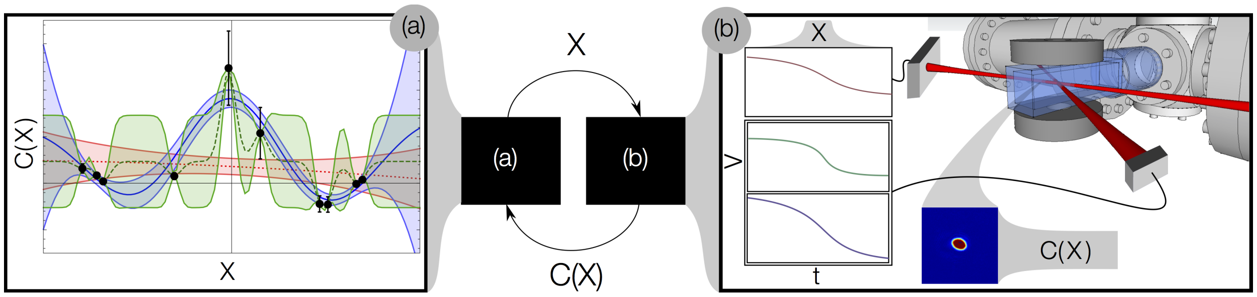


Fig. 1: M-LOOP in control of an ultra-cold atom experiment. M-LOOP was able to find an optimal set of ramps to evaporatively cool a thermal gas and form a Bose-Einstein Condensate.

Using M-LOOP is simple, once the parameters of your experiment is computer controlled, all you need to do is determine a cost function that quantifies the performance of an experiment after a single run. You can then hand over control of the experiment to M-LOOP which will find a global optimal set of parameters that minimize the cost function, by performing a few experiments and testing different parameters. M-LOOP uses machine-learning to predict how the parameters of the experiment relate to the cost, it uses this model to pick the next best parameters to test to find an optimum as quickly as possible.

M-LOOP not only finds an optimal set of parameters for the experiment it also provides a model of how the parameters are related to the costs which can be used to improve the experiment.

If you use M-LOOP please cite our publication where we first used the package to optimize the production of a Bose-Einstein Condensate:

Fast Machine-Learning Online Optimization of Ultra-Cold-Atom Experiments. *Scientific Reports* **6**, 25890 (2016). DOI: Link 10.1038/srep25890

http://www.nature.com/articles/srep25890

CHAPTER 1

Quick Start

To get M-LOOP running follow the *Installation* instructions and *Tutorials*.

CHAPTER 2

Contents

2.1 Installation

M-LOOP is available on PyPI and can be installed with your favorite package manager; simply search for 'M-LOOP' and install. However, if you want the latest features and a local copy of the examples you should install M-LOOP using the source code from the Link GitHub. Detailed installation instruction are provided below.

The installation process involves three steps.

1. Get a Python distribution with the standard scientific packages. We recommend installing *Anaconda*.
2. Install the latest release of *M-LOOP*.
3. (Optional) *Test* your M-LOOP install.

If you are having any trouble with the installation you may need to check your the *package dependencies* have been correctly installed. If you ares still having trouble, you can Link submit an issue to the GitHub.

2.1.1 Anaconda

We recommend installing Anaconda to get a python environment with all the required scientific packages. The Anaconda distribution is available here:

https://www.continuum.io/downloads

Follow the installation instructions they provide.

M-LOOP is targeted at python 3 but also supports 2. Please use python 3 if you do not have a reason to use 2, see *Python 3 vs 2* for details.

2.1.2 M-LOOP

You have two options when installing M-LOOP, you can perform a basic installation of the last release with pip or you can install from source to get the latest features. We recommend installing from source so you can test your installation, see all the examples and get the most recent bug fixes.

Installing from source

M-LOOP can be installed from the latest source code with three commands:

```
git clone git://github.com/michaelhush/M-LOOP.git
cd ./M-LOOP
python setup.py develop
```

The first command downloads the latest source code for M-LOOP from GitHub into the current directory, the second moves into the M-LOOP source directory, and the third link builds the package and creates a link from you python package to the source. If you are using linux or MacOS you may need admin privileges to run the setup script.

At any time you can update M-LOOP to the latest version from GitHub by running the command:

```
git pull origin master
```

in the M-LOOP directory.

Installing with pip

M-LOOP can be installed with pip with a single command:

```
pip install M-LOOP
```

If you are using linux or MacOS you may need admin privileges to run the command. To update M-LOOP to the latest version use:

```
pip install M-LOOP --upgrade
```

2.1.3 Testing

If you have installed from source, to test you installation use the command:

```
python setup.py test
```

In the M-LOOP source code directory. The tests should take around five minutes to complete. If you find a error please consider *Contributing* to the project and report a bug on the GitHub.

If you installed M-LOOP using pip, you will not need to test your installation.

2.1.4 Dependencies

M-LOOP requires the following packages to run correctly.

Package	Version
docutils	>=0.3
matplotlib	>=1.5
numpy	>=1.11
pip	>=7.0
pytest	>=2.9
setuptools	>=26
scikit-learn	>=0.18
scipy	>=0.17

These packages should be automatically installed by pip or the script setup.py when you install M-LOOP.

However, if you are using Anaconda some packages that are managed by the conda command may not be correctly updated, even if your installation passes all the tests. In this case, you will have to update these packages manually. You can check what packages you have installed and their version with the command:

```
conda list
```

To install a package that is missing, say for example pytest, use the command:

```
conda install pytest
```

To update a package to the latest version, say for example scikit-learn, use the command:

```
conda update scikit-learn
```

Once you install and update all the required packages with conda M-LOOP should run correctly.

2.1.5 Documentation

The latest documentation will always be available here online. If you would also like a local copy of the documentation, and you have downloaded the source code, enter the docs folder and use the command:

```
make html
```

Which will generate the documentation in docs/_build/html.

2.1.6 Python 3 vs 2

M-LOOP is developed in python 3 and it gets the best performance in this environment. This is primarily because other packages that M-LOOP uses, like numpy, run fastest in python 3. The tests typically take about 20% longer to complete in python 2 than 3.

If you have a specific reason to stay in a python 2 environment (you may use other packages which are not python 3 compatible) then you can still use M-LOOP without upgrading to 3. However, if you do not have a specific reason to stay with python 2, it is highly recommended you use the latest python 3 package.

2.2 Tutorials

Here we provide some tutorials on how to use M-LOOP. M-LOOP is flexible and can be customized with a variety of *options* and *interfaces*. Here we provide some basic tutorials to get you up and started as quick as possible.

There are two different approaches to using M-LOOP:

1. You can execute M-LOOP from a command line (or shell) and configure it using a text file.
2. You can use M-LOOP as a *python API*.

If you have a standard experiment, that is operated by LabVIEW, Simulink or some other method, then you should use option 1 and follow the *first tutorial*. If your experiment is operated using python, you should consider using option 2 as it will give you more flexibility and control, in which case, look at the *second tutorial*.

2.2.1 Standard experiment

The basic operation of M-LOOP is sketched below.

There are three stages:

1. M-LOOP is started with the command:

   ```
   M-LOOP
   ```

   M-LOOP first looks for the configuration file *exp_config.txt*, which contains options like the number of parameters and their limits, in the folder it is executed, then starts the optimization process.

2. M-LOOP controls and optimizes the experiment by exchanging files written to disk. M-LOOP produces a file called *exp_input.txt* which contains a variable params with the next parameters to be run by the experiment. The experiment is expected to run an experiment with these parameters and measure the resultant cost. The experiment should then write the file *exp_output.txt* which contains at least the variable cost which quantifies the performance of that experimental run, and optionally, the variables uncer (for uncertainty) and bad (if the run failed). This process is repeated many times until the halting condition is met.

3. Once the optimization process is complete, M-LOOP prints to the console the parameters and cost of the best run performed during the experiment, and a prediction of what the optimal parameters (with the corresponding predicted cost and uncertainty). M-LOOP also produces a set of plots that allow the user to visualize the optimization process and cost landscape. During operation and at the end M-LOOP write three files to disk:

   - *M-LOOP_[datetime].log* a log of the console output and other debugging information during the run.
   - *controller_archive_[datetime].txt* an archive of all the experimental data recorded and the results.
   - *learner_archive_[datetime].txt* an archive of the model created by the machine learner of the experiment.

In what follows we will unpack this process and give details on how to configure and run M-LOOP.

Launching M-LOOP

Launching M-LOOP is performed by executing the command M-LOOP on the console. You can also provide the name of your configuration file if you do not want to use the default with the command:

```
M-LOOP -c [config_filename]
```

Configuration File

The configuration file contains a list of options and settings for the optimization run. Each option must be started on a new line and formatted as:

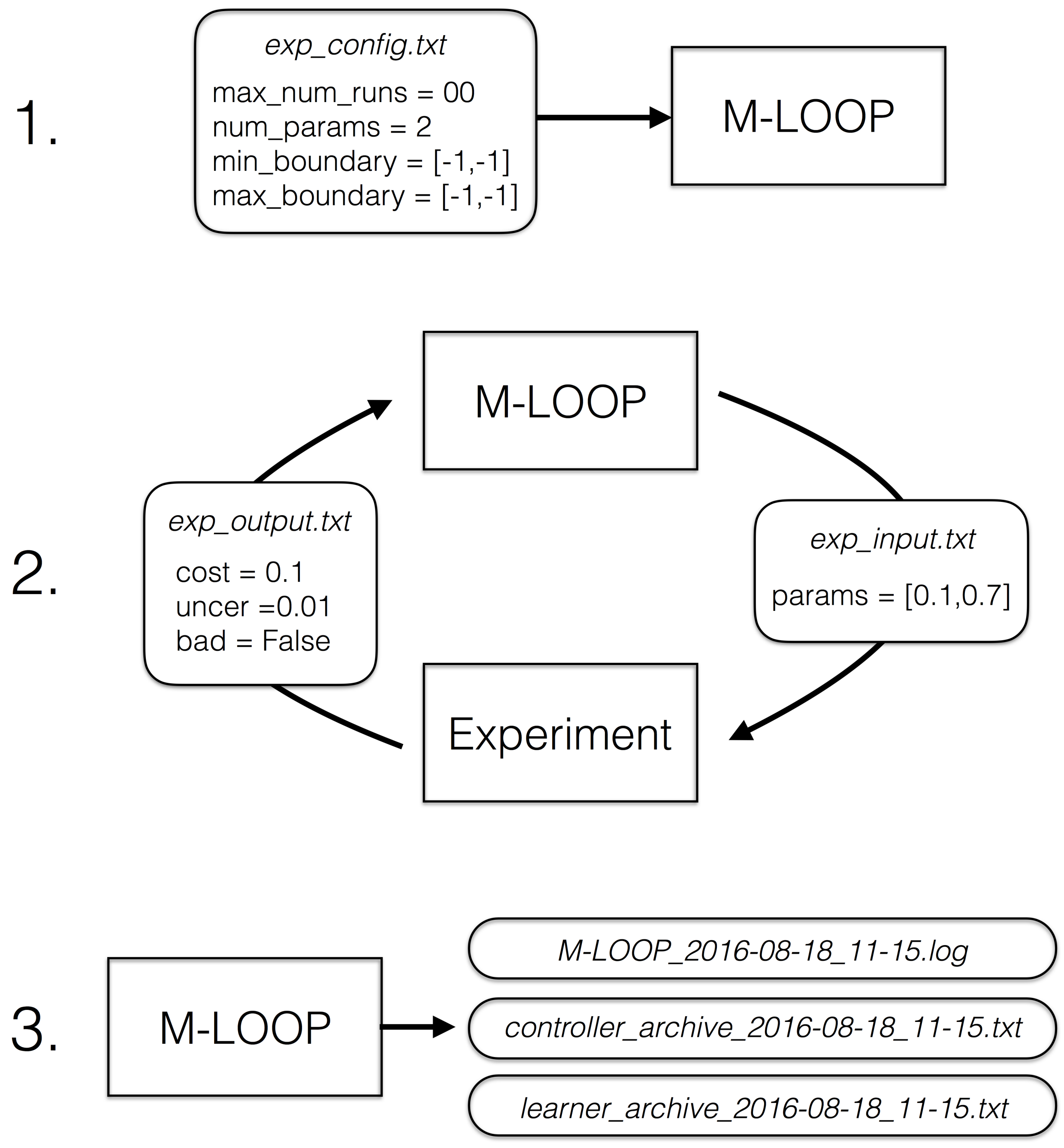
```
[keyword] = [value]
```

You can add comments to your file using #, everything past # will be ignored. Examples of relevant keywords and syntax for the values is provided in *Examples* and a comprehensive list of options is described in *Examples*. The values should be formatted with python syntax, strings should be surrounded with single or double quotes and arrays of values can be surrounded with square brackets/parentheses with numbers separated with commas. In this tutorial we will examine the example file *tutoral_config.txt*:

```
#Tutorial Config
#---------------

#Parameter settings
```

(continues on next page)


1.
exp_config.txt
max_num_runs = 00
num_params = 2
min_boundary = [-1,-1]
max_boundary = [-1,-1]
M-LOOP
2.
M-LOOP
exp_output.txt
cost = 0.1
uncer =0.01
bad = False
exp_input.txt
params = [0.1,0.7]
Experiment
3.
M-LOOP
M-LOOP_2016-08-18_11-15.log
controller_archive_2016-08-18_11-15.txt
learner_archive_2016-08-18_11-15.txt

(continued from previous page)

```
num_params = 2                #number of parameters
min_boundary = [-1,-1]        #minimum boundary
max_boundary = [1,1]          #maximum boundary
first_params = [0.5,0.5]      #first parameters to try
trust_region = 0.4           #maximum % move distance from best params

#Halting conditions
max_num_runs = 1000                        #maximum number of runs
max_num_runs_without_better_params = 50    #maximum number of runs without finding␣
↪better parameters
target_cost = 0.01                         #optimization halts when a cost below this␣
↪target is found

#Learner options
cost_has_noise = True      #whether the cost are corrupted by noise or not

#Timing options
no_delay = True            #wait for learner to make generate new parameters or use␣
↪training algorithms

#File format options
interface_file_type = 'txt'            #file types of *exp_input.mat* and *exp_output.
↪mat*
controller_archive_file_type = 'mat'   #file type of the controller archive
learner_archive_file_type = 'pkl'      #file type of the learner archive

#Visualizations
visualizations = True
```

We will now explain the options in each of their groups. In almost all cases you will only need to the parameters settings and halting conditions, but we have also described a few of the most commonly used extra options.

Parameter settings

The number of parameters and their limits is defined with three keywords:

```
num_params = 2
min_boundary = [-1,-1]
max_boundary = [1,1]
```

num_params defines the number of parameters, min_boundary defines the minimum value each of the parameters can take and max_boundary defines the maximum value each parameter can take. Here there are two value which each must be between -1 and 1.

first_parameters defines the first parameters the learner will try. You only need to set this if you have a safe set of parameters you want the experiment to start with. Just delete this keyword if any set of parameters in the boundaries will work.

trust_region defines the maximum change allowed in the parameters from the best parameters found so far. In the current example the region size is 2 by 2, with a trust region of 40% thus the maximum allowed change for the second run will be [0 +/- 0.8, 0 +/- 0.8]. This is only needed if your experiment produces bad results when the parameters are changes significantly between runs. Simply delete this keyword if your experiment works with any set of parameters within the boundaries.

Halting conditions

The halting conditions define when the simulation will stop. We present three options here:

```
max_num_runs = 100
max_num_runs_without_better_params = 10
target_cost = 0.1
first_params = [0.5,0.5]
trust_region = 0.4
```

max_num_runs is the maximum number of runs that the optimization algorithm is allowed to run. max_num_runs_without_better_params is the maximum number of runs allowed before a lower cost and better parameters is found. Finally, when target_cost is set, if a run produces a cost that is less than this value the optimization process will stop.

When multiple halting conditions are set, the optimization process will halt when any one of them is met.

If you do not have any prior knowledge of the problem use only the keyword max_num_runs and set it to the highest value you can wait for. If you have some knowledge about what the minimum attainable cost is or there is some cost threshold you need to achieve, you might want to set the target_cost. max_num_runs_without_better_params is useful if you want to let the optimization algorithm run as long as it needs until there is a good chance the global optimum has been found.

If you do not want one of the halting conditions, simply delete it from your file. For example if you just wanted the algorithm to search as long as it can until it found a global minimum you could set:

```
max_num_runs_without_better_params = 10
```

Learner Options

There are many learner specific options (and different learner algorithms) described in *Examples*. Here we just present a common one:

```
cost_has_noise = True
```

If the cost you provide has noise in it, meaning your the cost you calculate would fluctuate if you did multiple experiments with the same parameters, then set this flag to True. If the costs your provide have no noise then set this flag to False. M-LOOP will automatically determine if the costs have noise in them or not, so if you are unsure, just delete this keyword and it will use the default value of True.

Timing options

M-LOOP learns how the experiment works by fitting the parameters and costs using a gaussian process. This learning process can take some time. If M-LOOP is asked for new parameters before it has time to generate a new prediction, it will use the training algorithm to provide a new set of parameters to test. This allows for an experiment to be run while the learner is still thinking. The training algorithm by default is differential evolution, this algorithm is also used to do the first initial set of experiments which are then used to train M-LOOP. If you would prefer M-LOOP waits for the learner to come up with its best prediction before running another experiment you can change this behavior with the option:

```
no_delay = True
```

Set no_delay to true to ensure there is no pauses between experiments and set it to false if you to give M-LOOP to have the time to come up with its most informed choice. Sometimes doing fewer more intelligent experiments will lead to an optimal quicker than many quick unintelligent experiments. You can delete the keyword if you are unsure and it will default to True.

File format options

You can set the file formats for the archives produced at the end and the files exchanged with the experiment with the options:

```
interface_file_type = 'txt'
controller_archive_file_type = 'mat'
learner_archive_file_type = 'pkl'
```

interface_file_type controls the file format for the files exchanged with the experiment. controller_archive_file_type and learner_archive_file_type control the format of the respective archives.

There are three file formats currently available: 'mat' is for MATLAB readable files, 'pkl' if for python binary archives created using the pickle package, and 'txt' human readable text files. For more details on these formats see *Data*.

Visualization

By default M-LOOP will display a set of plots that allow the user to visualize the optimization process and the cost landscape. To change this behavior use the option:

```
visualizations = True
```

Set it to false to turn the visualizations off. For more details see *Visualizations*.

Interface

There are many options of how to connect M-LOOP to your experiment. We consider the most generic method, writing and reading files to disk. For other options see *Interfaces*. If you design a bespoke interface for your experiment please consider *Contributing* to the project by sharing your method with other users.

The file interface works under the assumption that you experiment follows the following algorithm.

1. Wait for the file *exp_input.txt* to be made on the disk in the same folder M-LOOP is run.
2. Read the parameters for the next experiment from the file (named params).
3. Delete the file *exp_input.txt*.
4. Run the experiment with the parameters provided and calculate a cost, and optionally the uncertainty.
5. Write the cost to the file *exp_output.txt*. Go back to step 1.

It is important you delete the file *exp_input.txt* after reading it, since it is used to as an indicator for the next experiment to run.

When writing the file *exp_output.txt* there are three keywords and values you can include in your file, for example after the first run your experiment may produce the following:

```
cost = 0.5
uncer = 0.01
bad = false
```

cost refers to the cost calculated from the experimental data. uncer, is optional, and refers to the uncertainty in the cost measurement made. Note, M-LOOP by default assumes there is some noise corrupting costs, which is fitted and compensated for. Hence, if there is some noise in your costs which you are unable to predict from a single measurement, do not worry, you do not have to estimate uncer, you can just leave it out. Lastly bad can be used to indicate an experiment failed and was not able to produce a cost. If the experiment worked set bad = false and if it failed set bad = true.

Note you do not have to include all of the keywords, you must provide at least a cost or the bad keyword set to true. For example a successful run can simply be:

```
cost = 0.3
```

and failed experiment can be as simple as:

```
bad = True
```

Once the *exp_output.txt* has been written to disk, M-LOOP will read it and delete it.

Parameters and cost function

Choosing the right parameterization of your experiment and cost function will be an important part of getting great results.

If you have time dependent functions in your experiment you will need to choose a parametrization of these function before interfacing them with M-LOOP. M-LOOP will take more time and experiments to find an optimum, given more parameters. But if you provide too few parameters, you may not be able to achieve your cost target.

Fortunately, the visualizations provided after the optimization will help you determine which parameters contributed the most to the optimization process. Try with whatever parameterization is convenient to start and use the data produced afterwards to guide you on how to better improve the parametrization of your experiment.

Picking the right cost function from experimental observables will also be important. M-LOOP will always find a global optimal as quick as it can, but if you have a poorly chosen cost function, the global optimal may not what you really wanted to optimize. Make sure you pick a cost function that will uniquely produce the result you want. Again, do not be afraid to experiment and use the data produced by the optimization runs to improve the cost function you are using.

Have a look at our paper on using M-LOOP to create a Bose-Einstein Condensate for an example of choosing a parametrization and cost function for an experiment.

Results

Once M-LOOP has completed the optimization, it will output results in several ways.

M-LOOP will print results to the console. It will give the parameters of the experimental run that produced the lowest cost. It will also provide a set of parameters which are predicted to be produce the lowest average cost. If there is no noise in the costs your experiment produced, then the best parameters and predicted best parameters will be the same. If there was some noise your costs then it is possible that there will be a difference between the two. This is because the noise might have resulted with a set of experimental parameters that produced a lower cost due to a random fluke. The real optimal parameters that correspond to the minimum average cost are the predicted best parameters. In general, use the predicted best parameters (when provided) as the final result of the experiment.

M-LOOP will produce an archive for the controller and machine learner. The controller archive contains all the data gathered during the experimental run and also other configuration details set by the user. By default it will be a 'txt' file which is human readable. If the meaning of a keyword and its associated data in the file is unclear, just search the documentation with the keyword to find a description. The learner archive contains a model of the experiment produced by the machine learner algorithm, which is currently a gaussian process. By default it will also be a 'txt' file. For more detail on these files see *Data*.

M-LOOP, by default, will produce a set of visualizations. These plots show the optimizations process over time and also predictions made by the learner of the cost landscape. For more details on these visualizations and their interpretation see *Visualizations*.

2.2.2 Python controlled experiment

If you have an experiment that is already under python control you can use M-LOOP as an API. Below we go over the example python script *python_controlled_experiment.py* you should also read over the *first tutorial* to get a general idea of how M-LOOP works.

When integrating M-LOOP into your laboratory remember that it will be controlling you experiment, not vice versa. Hence, at the top level of your python script you will execute M-LOOP which will then call on your experiment when needed. Your experiment will not be making calls of M-LOOP.

An example script for a python controlled experiment is given in the examples folder called *python_controlled_experiment.py*, which is copied below:

```
#Imports for python 2 compatibility
from __future__ import absolute_import, division, print_function
__metaclass__ = type

#Imports for M-LOOP
import mloop.interfaces as mli
import mloop.controllers as mlc
import mloop.visualizations as mlv

#Other imports
import numpy as np
import time

#Declare your custom class that inherits from the Interface class
class CustomInterface(mli.Interface):

        #Initialization of the interface, including this method is optional
        def __init__(self):
                #You must include the super command to call the parent class,␣
↪Interface, constructor
                super(CustomInterface,self).__init__()

                #Attributes of the interface can be added here
                #If you want to pre-calculate any variables etc. this is the place to␣
↪do it
                #In this example we will just define the location of the minimum
                self.minimum_params = np.array([0,0.1,-0.1])

        #You must include the get_next_cost_dict method in your class
        #this method is called whenever M-LOOP wants to run an experiment
        def get_next_cost_dict(self,params_dict):

                #Get parameters from the provided dictionary
                params = params_dict['params']

                #Here you can include the code to run your experiment given a␣
↪particular set of parameters
```

(continues on next page)

(continued from previous page)

```
                #In this example we will just evaluate a sum of sinc functions
                cost = -np.sum(np.sinc(params - self.minimum_params))
                #There is no uncertainty in our result
                uncer = 0
                #The evaluation will always be a success
                bad = False
                #Add a small time delay to mimic a real experiment
                time.sleep(1)

                #The cost, uncertainty and bad boolean must all be returned as a␣
↪dictionary
                #You can include other variables you want to record as well if you␣
↪want
                cost_dict = {'cost':cost, 'uncer':uncer, 'bad':bad}
                return cost_dict

def main():
        #M-LOOP can be run with three commands

        #First create your interface
        interface = CustomInterface()
        #Next create the controller, provide it with your controller and any options␣
↪you want to set
        controller = mlc.create_controller(interface, max_num_runs = 1000, target_
↪cost = -2.99, num_params = 3, min_boundary = [-2,-2,-2], max_boundary = [2,2,2])
        #To run M-LOOP and find the optimal parameters just use the controller method␣
↪optimize
        controller.optimize()

        #The results of the optimization will be saved to files and can also be␣
↪accessed as attributes of the controller.
        print('Best parameters found:')
        print(controller.best_params)

        #You can also run the default sets of visualizations for the controller with␣
↪one command
        mlv.show_all_default_visualizations(controller)


#Ensures main is run when this code is run as a script
if __name__ == '__main__':
        main()
```

Each part of the code is explained in the following sections.

Imports

The start of the script imports the libraries that are necessary for M-LOOP to work:

```
#Imports for python 2 compatibility
from __future__ import absolute_import, division, print_function
__metaclass__ = type

#Imports for M-LOOP
import mloop.interfaces as mli
```

(continues on next page)

(continued from previous page)

```
import mloop.controllers as mlc
import mloop.visualizations as mlv

#Other imports
import numpy as np
import time
```

The first group of imports are just for python 2 compatibility. M-LOOP is targeted at python3, but has been designed to be bilingual. These imports ensure backward compatibility.

The second group of imports are the most important modules M-LOOP needs to run. The interfaces and controllers modules are essential, while the visualizations module is only needed if you want to view your data afterwards.

Lastly, you can add any other imports you may need.

Custom Interface

M-LOOP takes an object oriented approach to controlling the experiment. This is different than the functional approach taken by other optimization packages, like scipy. When using M-LOOP you must make your own class that inherits from the Interface class in M-LOOP. This class must implement a method called *get_next_cost_dict* that takes a set of parameters, runs your experiment and then returns the appropriate cost and uncertainty.

An example of the simplest implementation of a custom interface is provided below

```
#Declare your custom class that inherits from the Interface class
class SimpleInterface(mli.Interface):

        #the method that runs the experiment given a set of parameters and returns a␣
↪cost
        def get_next_cost_dict(self,params_dict):

                #The parameters come in a dictionary and are provided in a numpy array
                params = params_dict['params']pre-calculate

                #Here you can include the code to run your experiment given a␣
↪particular set of parameters
                #For this example we just evaluate a simple function
                cost = np.sum(params**2)
                uncer = 0
                bad = False

                #The cost, uncertainty and bad boolean must all be returned as a␣
↪dictionary
                cost_dict = {'cost':cost, 'uncer':uncer, 'bad':bad}
                return cost_dict
```

The code above defines a new class that inherits from the Interface class in M-LOOP. Note this code is different to the example above, we will consider this later. It is slightly more complicated than just defining a method, however there is a lot more flexibility when taking this approach. You should put the code you use to run your experiment in the *get_next_cost_dict* method. This method is executed by the interface whenever M-LOOP wants a cost corresponding to a set of parameters.

When you actually run M-LOOP you will need to make an instance of your interface. To make an instance of the class above you would use:

```
interface = SimpleInterface()
```

This interface is then provided to the controller, which is discussed in the next section.

Dictionaries are used for both input and output of the method, to give the user flexibility. For example, if you had a bad run, you do not have to return a cost and uncertainty, you can just return a dictionary with bad set to True:

```
cost_dict = {'bad':True}
return cost_dict
```

By taking an object oriented approach, M-LOOP can provide a lot more flexibility when controlling your experiment. For example if you wish to start up your experiment or perform some initial numerical analysis you can add a customized constructor or __init__ method for the class. We consider this in the main example:

```
class CustomInterface(mli.Interface):

        #Initialization of the interface, including this method is optional
        def __init__(self):
                #You must include the super command to call the parent class,␣
↪Interface, constructor
                super(CustomInterface,self).__init__()

                #Attributes of the interface can be added here
                #If you want to pre-calculate any variables etc. this is the place to␣
↪do it
                #In this example we will just define the location of the minimum
                self.minimum_params = np.array([0,0.1,-0.1])

        #You must include the get_next_cost_dict method in your class
        #this method is called whenever M-LOOP wants to run an experiment
        def get_next_cost_dict(self,params_dict):

                #Get parameters from the provided dictionary
                params = params_dict['params']

                #Here you can include the code to run your experiment given a␣
↪particular set of parameters
                #In this example we will just evaluate a sum of sinc functions
                cost = -np.sum(np.sinc(params - self.minimum_params))
                #There is no uncertainty in our result
                uncer = 0
                #The evaluation will always be a success
                bad = False
                #Add a small time delay to mimic a real experiment
                time.sleep(1)

                #The cost, uncertainty and bad boolean must all be returned as a␣
↪dictionary
                #You can include other variables you want to record as well if you␣
↪want
                cost_dict = {'cost':cost, 'uncer':uncer, 'bad':bad}
                return cost_dict
```

In this code snippet we also implement a constructor. Here we just define a numpy array which defines the minimum_parameter values. We can call this variable whenever we need in the *get_next_cost_dict method*. You can also define your own custom methods in your interface or even inherit from other classes.

Once you have implemented your own Interface running M-LOOP can be done in three lines.

Running M-LOOP

Once you have made your interface class running M-LOOP can be as simple as three lines. In the example script M-LOOP is run in the main method:

```
def main():
        #M-LOOP can be run with three commands

        #First create your interface
        interface = CustomInterface()
        #Next create the controller, provide it with your controller and any options␣
↪you want to set
        controller = mlc.create_controller(interface, max_num_runs = 1000, target_
↪cost = -2.99, num_params = 3, min_boundary = [-2,-2,-2], max_boundary = [2,2,2])
        #To run M-LOOP and find the optimal parameters just use the controller method␣
↪optimize
        controller.optimize()
```

In the code snippet we first make an instance of our custom interface class called interface. We then create an instance of a controller. The controller will run the experiment and perform the optimization. You must provide the controller with the interface and any of the M-LOOP options you would normally provide in the configuration file. In this case we give five options, which do the following:

1. *max_num_runs* = *1000* sets the maximum number of runs to be 1000.
2. *target_cost* = *-2.99* sets a cost that M-LOOP will halt at once it has been reached.
3. *num_params* = *3* sets the number of parameters to be 3.
4. *min_boundary* = *[-2,-2,-2]* defines the minimum values of each of the parameters.
5. *max_boundary* = *[2,2,2]* defines the maximum values of each of the parameters.

There are many other options you can use. Have a look at *Configuration File* for a detailed introduction into all the important configuration options. Remember you can include any option you would include in a configuration file as keywords for the controller. For more options you should look at all the config files in *Examples*, or for a comprehensive list look at the *M-LOOP API*.

Once you have created your interface and controller you can run M-LOOP by calling the optimize method of the controller. So in summary M-LOOP is executed in three lines:

```
interface = CustomInterface()
controller = mlc.create_controller(interface, [options])
controller.optimize()
```

Results

The results will be displayed on the console and also saved in a set of files. Have a read over *Results* for more details on the results displayed and saved. Also read *Data* for more details on data formats and how it is stored.

Within the python environment you can also access the results as attributes of the controller after it has finished optimization. The example includes a simple demonstration of this:

```
#The results of the optimization will be saved to files and can also be accessed as␣
↪attributes of the controller.
print('Best parameters found:')
print(controller.best_params)
```

All of the results saved in the controller archive can be directly accessed as attributes of the controller object. For a comprehensive list of the attributes of the controller generated after an optimization run see the *M-LOOP API*.

Visualizations

For each controller there is normally a default set of visualizations available. The visualizations for the Gaussian Process, the default optimization algorithm, is described in *Visualizations*. Visualizations can be called through the visualization module. The example includes a simple demonstration of this:

```
#You can also run the default sets of visualizations for the controller with one␣
↪command
mlv.show_all_default_visualizations(controller)
```

This code snippet will display all the visualizations available for that controller. There are many other visualization methods and options available that let you control which plots are displayed and when, see the *M-LOOP API* for details.

2.3 Interfaces

Currently M-LOOP supports three ways to interface your experiment

1. File interface where parameters and costs are exchanged between the experiment and M-LOOP through files written to disk. This approach is described in a *tutorial*.
2. Shell interface where parameters and costs are exchanged between the experiment and M-LOOP through information piped through a shell (or command line). This option should be considered if you can execute your experiment using a command from a shell.
3. Implementing your own interface through the M-LOOP python API.

Each of these options is described below. If you have any suggestions for interfaces please consider *Contributing* to the project.

2.3.1 File interface

The simplest method to connect your experiment to M-LOOP is with the file interface where data is exchanged by writing files to disk. To use this interface you can include the option:

```
interface='file'
```

in your configuration file. The file interface happens to be the default, so this is not necessary.

The file interface works under the assumption that you experiment follows the following algorithm.

1. Wait for the file *exp_input.txt* to be made on the disk in the same folder M-LOOP is run.
2. Read the parameters for the next experiment from the file (named params).
3. Delete the file *exp_input.txt*.
4. Run the experiment with the parameters provided and calculate a cost, and optionally the uncertainty.
5. Write the cost to the file *exp_output.txt*. Go back to step 1.

It is important you delete the file *exp_input.txt* after reading it, since it is used to as an indicator for the next experiment to run.

When writing the file *exp_output.txt* there are three keywords and values you can include in your file, for example after the first run your experiment may produce the following:

```
cost = 0.5
uncer = 0.01
bad = false
```

cost refers to the cost calculated from the experimental data. uncer, is optional, and refers to the uncertainty in the cost measurement made. Note, M-LOOP by default assumes there is some noise corrupting costs, which is fitted and compensated for. Hence, if there is some noise in your costs which you are unable to predict from a single measurement, do not worry, you do not have to estimate uncer, you can just leave it out. Lastly bad can be used to indicate an experiment failed and was not able to produce a cost. If the experiment worked set bad = false and if it failed set bad = true.

Note you do not have to include all of the keywords, you must provide at least a cost or the bad keyword set to false. For example a successful run can simply be:

```
cost = 0.3
```

and failed experiment can be as simple as:

```
bad = True
```

Once the *exp_output.txt* has been written to disk, M-LOOP will read it and delete it.

2.3.2 Shell interface

The shell interface is used when experiments can be run from a command in a shell. M-LOOP will still need to be configured and executed in the same manner described for a file interface as describe in *tutorial*. The only difference is how M-LOOP starts the experiment and reads data. To use this interface you must include the following options:

```
interface_type='shell'
command='./run_exp'
params_args_type='direct'
```

in the configuration file. The interface keyword simply indicates that you want M-LOOP to operate the experiment through the shell. The other two keywords need to be customized to your needs.

The command keyword should be provided with the command on the shell that runs the experiment. In the example above the executable would be *run_exp*. Note M-LOOP will try and execute the command in the folder that you run M-LOOP from, if this causes trouble you should just the absolute address of your executable. Your command can be more complicated than a single work, for example if you wanted to include some options like './run_exp –verbose -U' this would also be acceptable.

The params_args_type keyword controls how M-LOOP delivers the parameters to the executable. If you use the 'direct' option the parameters will just be fed directly to the experiment as arguments. For example if the command was ./run_exp and the parameters to test next were 1.3, -23 and 12, M-LOOP would execute the following command:

```
./run_exp 1.3 -23 12
```

the other params_args_type option is 'named' in this case each parameter is fed to the experiment as a named option. Given the same parameters as before, M-LOOP would execute the command:

```
./run_exp --param1 1.3 --param2 -23 --param3 12
```

After the experiment has run and a cost (and uncertainty or bad value) has been found they must be provided back to M-LOOP through the shell. For example if you experiment completed with a cost 1.3, uncertainty 0.1 you need to program your executable to print the following to the shell:

```
M-LOOP_start
cost = 1.3
uncer = 0.1
M-LOOP_end
```

You can also output other information to the shell and split up the information you provide to M-LOOP if you wish. The following output would also valid.

> Running experiment… Experiment complete. Checking it was valid… It worked. M-LOOP_start bad = False M-LOOP_end Calculating cost… Was 3.2. M-LOOP_start cost = 3.2 M-LOOP_end

2.3.3 Python interfaces

If your experiment is controlled in python you can use M-LOOP as an API in your own custom python script. In this case you must create your own implementation of the abstract interface class to control the experiment. This is explained in detail in the *tutorial for python controlled experiments*.

2.4 Data

M-LOOP saves all data produced by the experiment in archives which are saved to disk during and after the optimization run. The archives also contain information derived from the data, including the machine learning model for how the experiment works. Here we explain how to interpret the file archives.

2.4.1 File Formats

M-LOOP currently supports three file formats for all file input and output.

- 'txt' text files: Human readable text files. This is the default file format for all outputs. The advantage of text files is they are easy to read, and there will be no format compatibility issues in the future. However, there will be some loss of precision in your data. To ensure you keep all significant figure you may want to use 'pkl' or 'mat'.
- 'mat' MATLAB files: Matlab files that can be opened and written with MATLAB or numpy.
- 'pkl' pickle files: a serialization of a python dictionary made with *pickle <https://docs.python.org/3/library/pickle.html>*. Your data can be retrieved from this dictionary using the appropriate keywords.

2.4.2 File Keywords

The archives contain a set of keywords/variable names with associated data. The quickest way to understand what the values mean for a particular keyword is to search the documentation for a description.

For a comprehensive list of all the keywords looks at the attributes described in the API.

For the controller archive see *controllers*.

For the learner archive see *learners*. The generic keywords are described in the class Learner, with learner specific options described in the derived classes, for example GaussianProcessLearner.

2.4.3 Converting files

If for whatever reason you want to convert files between the formats you can do so using the utilities module of M-LOOP. For example the following python code will convert the file controller_archive_2016-08-18_12-18.pkl from a 'pkl' file to a 'mat' file:

```
import mloop.utilities as mlu

saved_dict = mlu.get_dict_from_file('./M-LOOP_archives/controller_archive_2016-08-18_
↪12-18.pkl','pkl')
mlu.save_dict_to_file(saved_dict,'./M-LOOP_archives/controller_archive_2016-08-18_12-
↪18.mat','mat')
```

2.5 Visualizations

At the end of an optimization run a set of visualizations will be produce by default.

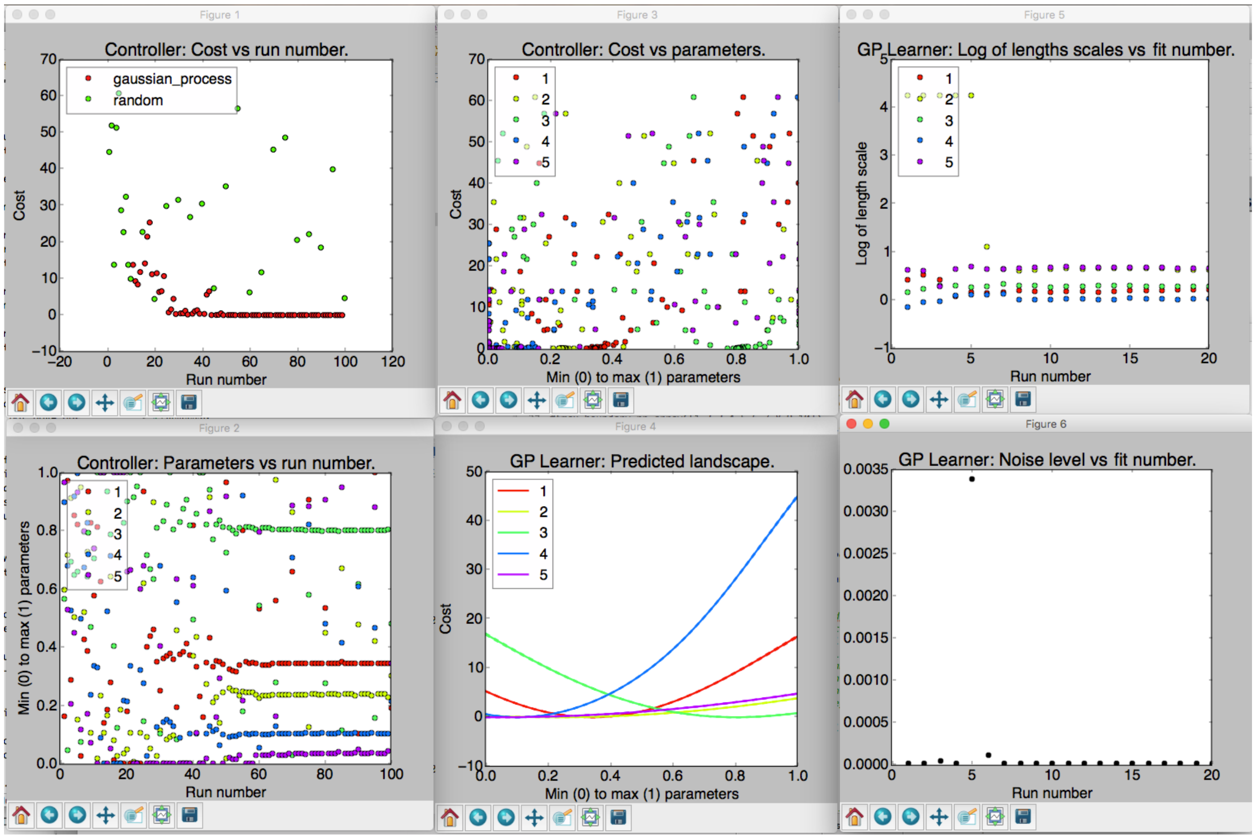


Fig. 1: An example of the six visualizations automatically produced when M-LOOP is run with the default controller, the Gaussian process machine learner.

The number of visualizations will depend on what controller you use. By default there should be six which are described below:

- **Controller: Cost vs run number.** Here the returned by the experiment versus run number is plotted. The legend shows what algorithm was used to generate the parameters tested by the experiment. If you use the Gaussian process, there will also be another algorithm used throughout the optimization algorithm in order to (a) ensure parameters are generated fast enough and (b) add new prior free data to ensure the Gaussian process converges to the correct model.

- **Controller: Parameters vs run number.** The parameters values are all plotted against the run number. Note the parameters will all be scaled between their minimum and maximum value. the legend indicates what color corresponds to what parameter.
- **Controller: Cost vs parameters.** The cost versus the parameters. Here each of the parameters tested are plotted against the cost they returned as a set. Again the parameter values are all scaled between their minimum and maximum values.
- **GP Learner: Predicted landscape.** 1D cross sections of the landscape about the best recorded cost are plotted against each parameter. The color of the cross section corresponds to the parameter that is varied in the cross section. This predicted landscape is generated by the model fit to the experiment by the Gaussian process. Be sure to check after an optimization run that all parameters contributed. If one parameter produces a flat cross section, it is most likely it did not have any influence on the final cost. You may want to remove it on the next optimization run.
- **GP Learner: Log of length scales vs fit number.** The Gaussian process fits a correlation length to each of the parameters in the experiment. Here we see a plot of the correlation lengths versus fit number. The last correlation lengths (highest fit number) is the most reliable values. Correlation lengths indicate how sensitive the cost is to changes in these parameters. If the correlation length is large, the parameter has a very little influence on the cost, if the correlation length is small, the parameter will have a very large influence on the cost. The correlation lengths are not precisely estimate. They should only be trusted accurate to +/- an order of magnitude. If a parameter has an extremely large value at the end of the optimization, say 5 or more, it is unlikely to have much affect on the cost and should be removed on the next optimization run.
- **GP Learner: Noise level vs fit number.** This is the estimated noise in the costs as a function of fit number. The most reliable estimate of the noise level will be the last value (highest fit number). The noise level is useful for quantifying the intrinsic noise and uncertainty in your cost value. Most other optimization algorithms will not provide this estimate. The noise level estimate may be helpful when isolating what part of your system can be optimized and what part is due to random fluctuations.

The plots which start with *Controller:* are generated from the controller archive, while plots that start with *Learner:* are generated from the learner archive.

2.5.1 Reproducing visualizations

If you have a controller and learner archive and would like to examine the visualizations again, it is best to do so using the *M-LOOP API*. For example the following code will plot the visualizations again from the files *controller_archive_2016-08-23_13-59.mat* and *learner_archive_2016-08-18_12-18.pkl*:

```
import mloop.visualizations as mlv
import matplotlib.pyplot as plt

mlv.configure_plots()
mlv.create_controller_visualizations('controller_archive_2016-08-23_13-59.mat',file_
↪type='mat')
mlv.create_gaussian_process_learner_visualizations('learner_archive_2016-08-18_12-18.
↪pkl',file_type='pkl')

plt.show()
```

2.6 Examples

M-LOOP includes a series of example configuration files for each of the controllers and interfaces. The examples can be found in examples folder. For some controllers there are two files, ones ending with *_basic_config* which includes the standard configuration options and *_complete_config* which include a comprehensive list of all the configuration options available.

The options available are also comprehensively documented in the *M-LOOP API* as keywords for each of the classes. However, the quickest and easiest way to learn what options are available, if you are not familiar with python, is to just look at the provided examples.

Each of the example files is used when running tests of M-LOOP. So please copy and modify them elsewhere if you use them as a starting point for your configuration file.

2.6.1 Interfaces

There are currently two interfaces supported: 'file' and 'shell'. You can specify which interface you want with the option:

```
interface_type = [name]
```

The default will be 'file'. The specific options for each of the interfaces are described below.

File Interface

The file interface exchanges information with the experiment by writing files to disk. You can change the names of the files used for the file interface and their type. The file interface options are described in *file_interface_config.txt*.

```
#File Interface Options
#----------------------

interface_type = 'file'                #The type of interface
interface_out_filename = 'exp_input'   #The filename of the file output by the␣
↪interface and input into the experiment
interface_in_filename  = 'exp_output'  #The filename o the file input into the␣
↪interface and output by the experiment
interface_file_type = 'txt'            #The file_type of both the input and output␣
↪files, can be 'txt', 'pkl' or 'mat'.
```

Shell Interface

The shell interface is for experiments that can be run through a command executed in a shell. Information is then piped between M-LOOP and the experiment through the shell. You can change the command to run the experiment and the way the parameters are formatted. The shell interface options are described in *shell_interface_config.txt*

```
#Command Line Interface Options
#------------------------------

interface_type = 'shell'                        #The type of interface
command = 'python shell_script.py'      #The command for the command line to␣
↪run the experiment to get a cost from the parameters
```

```
params_args_type = 'direct'              #The format of the parameters when␣
↪providing them on the command line. 'direct' simply appends them, e.g.␣
↪python shell_script.py 7 2 1, 'named' names each parameter, e.g. python␣
↪shell_script.py --param1 7 --param2 2 --param3 1
```

2.6.2 Controllers

There are currently three controller types supported: 'gaussian_process', 'random' and 'nelder_mead'. The default is 'gaussian_process'. You can set which interface you want to use with the option:

```
controller_type = [name]
```

Each of the controllers and their specific options are described below. There is also a set of common options shared by all controllers which is described in *controller_options.txt*. The common options include the parameter settings and the halting conditions.

```
#General Controller Options
#--------------------------

#Halting conditions
max_num_runs = 1000                       #number of planned runs
target_cost = 0.1                         #cost to beat
max_num_runs_without_better_params = 100  #max allowed number of runs between finding␣
↪better parameters

#Parameter controls
num_params = 2          #Number of parameters
min_boundary = [0,0]    #Minimum value for each parameter
max_boundary = [2,2]    #Maximum value for each parameter

#Filename related
controller_archive_filename = 'agogo'     #filename prefix for controller archive
controller_archive_file_type = 'mat'      #file_type for controller archive
learner_archive_filename = 'ogoga'        #filename prefix for learner archive
learner_archive_file_type = 'pkl'         #file_type for learner archive
archive_extra_dict = {'test':'this_is'}   #dictionary of any extra data to be put in␣
↪archive
```

Gaussian process

The Gaussian-process controller is the default controller and is the currently the most sophisticated machine learner algorithm. It uses a Link Gaussian process to develop a model for how the parameters relate to the measured cost, effectively creating a model for how the experiment operates. This model is then used when picking new points to test.

There are two example files for the Gaussian-process controller: *gaussian_process_simple_config.txt* which contains the basic options.

```
#Gaussian Process Basic Options
#------------------------------

#General options
max_num_runs = 100                #number of planned runs
target_cost = 0.1

#Gaussian process controller options
```

(continues on next page)

(continued from previous page)

```
controller_type = 'gaussian_process'   #name of controller to use
num_params = 3                         #number of parameters
min_boundary = [-0.8,-0.9,-1.1]        #minimum boundary
max_boundary = [0.8,0.9,1.1]           #maximum boundary
trust_region = 0.4                     #maximum % move distance from best params
cost_has_noise = False                 #whether cost function has noise
```

gaussian_process_complete_config.txt which contains a comprehensive list of options.

```
#Gaussian Process Complete Options
#---------------------------------

#General options
max_num_runs = 100               #number of planned runs
target_cost = 0.1                #cost to beat

#Gaussian process options
controller_type = 'gaussian_process'
num_params = 2                         #number of parameters
min_boundary = [-10.,-10.]             #minimum boundary
max_boundary = [10.,10.]               #maximum boundary
length_scale = [1.0]                   #initial lengths scales for GP
cost_has_noise = True                  #whether cost function has noise
noise_level = 0.1                      #initial noise level
update_hyperparameters = True          #whether noise level and lengths scales are␣
↪updated
trust_region = [5,5]                   #maximum move distance from best params
default_bad_cost = 10                  #default cost for bad run
default_bad_uncertainty = 1            #default uncertainty for bad run
learner_archive_filename = 'a_word'    #filename of gp archive
learner_archive_file_type = 'mat'      #file type of archive
predict_global_minima_at_end  = True   #find predicted global minima at end
no_delay = True                       #whether to wait for the GP to make predictions␣
↪or not. Default True (do not wait)

#Training source options
training_type = 'random'               #training type can be random or nelder_mead
first_params = [1.9,-1.0]              #first parameters to try in initial training
gp_training_filename = None            #filename for training from previous experiment
gp_training_file_type  = 'pkl'         #training data file type

#if you use nelder_mead for the initial training source see the␣
↪CompleteNelderMeadConfig.txt for options.
```

Differential evolution

The differential evolution (DE) controller uses a Link DE algorithm for optimization. DE is a type of evolutionary algorithm, and is historically the most commonly used in automated optimization. DE will eventually find a global solution, however it can take many experiments before it does so.

There are two example files for the differential evolution controller: *differential_evolution_simple_config.txt* which contains the basic options.

```
#Differential Evolution Basic Options
#------------------------------------
```

(continues on next page)

(continued from previous page)

```
#General options
max_num_runs = 500              #number of planned runs
target_cost = 0.1               #cost to beat

#Differential evolution controller options
controller_type = 'differential_evolution'
num_params = 1                  #number of parameters
min_boundary = [-4.8]           #minimum boundary
max_boundary = [10.0]           #maximum boundary
trust_region = 0.6              #maximum % move distance from best params
first_params = [5.3]            #first parameters to try
```

differential_evolution_complete_config.txt which contains a comprehensive list of options.

```
#Differential Evolution Complete Options
#---------------------------------------

#General options
max_num_runs = 500              #number of planned runs
target_cost = 0.1               #cost to beat

#Differential evolution controller options
controller_type = 'differential_evolution'
num_params = 2                  #number of parameters
min_boundary = [-1.2,-2]        #minimum boundary
max_boundary = [10.0,4]         #maximum boundary
trust_region = [3.2,3.1]        #maximum move distance from best params
first_params = None             #first parameters to try if None a random set␣
↪of parameters is chosen
evolution_strategy='best2'              #evolution strategy can be 'best1',␣
↪'best2', 'rand1' and 'rand2'. Best uses the best point, rand uses a random␣
↪one, the number indicates the number of directions added.
population_size=10                              #a multiplier for the␣
↪population size of a generation
mutation_scale=(0.4, 1.1)               #the minimum and maximum value for␣
↪the mutation scale factor. Each generation is randomly selected from this.␣
↪Each value must be between 0 and 2.
cross_over_probability=0.8      #the probability a parameter will be␣
↪resampled during a mutation in a new generation
restart_tolerance=0.02                  #the fraction the standard deviation␣
↪in the costs of the population must reduce from the initial sample, before␣
↪the search is restarted.
```

Nelder Mead

The Nelder Mead controller implements the Link Nelder-Mead method for optimization. You can control the starting point and size of the initial simplex of the method with the configuration file.

There are two example files for the Nelder-Mead controller: *nelder_mead_simple_config.txt* which contains the basic options.

```
#Nelder-Mead Basic Options
#--------------------------

#General options
max_num_runs = 100           #number of planned runs
target_cost = 0.1                #cost to beat

#Specific options
controller_type = 'nelder_mead'
num_params = 3                   #number of parameters
min_boundary = [-1,-1,-1]      #minimum boundary
max_boundary = [1,1,1]       #maximum boundary
initial_simplex_scale = 0.4      #initial size of simplex relative to the boundary␣
↪size.
```

nelder_mead_complete_config.txt which contains a comprehensive list of options.

```
#Nelder-Mead Complete Options
#-----------------------------

#General options
max_num_runs = 100                                  #number of planned runs
target_cost = 0.1                                   #cost to beat

#Specific options
controller_type = 'nelder_mead'
num_params = 5                                               #number of parameters
min_boundary = [-1.1,-1.2, -1.3, -1.4, -1.5]                 #minimum boundary
max_boundary = [1.1, 1.1, 1.1, 1.1, 1.1]                     #maximum boundary
initial_simplex_corner= [-0.21,-0.23,-0.24,-0.23,-0.25]      #initial corner of the␣
↪simplex
initial_simplex_displacements=[1,1,1,1,1]                    #initial displacements for␣
↪the N+1 (i this case 6) points of the simplex
```

Random

The random optimization algorithm picks parameters randomly from a uniform distribution from within the parameter bounds or trust region.

There are two example files for the random controller: *random_simple_config.txt* which contains the basic options.

```
#Random Basic Options
#--------------------

#General options
max_num_runs = 10            #number of planned runs

#Random controller options
controller_type = 'random'
num_params = 1                   #number of parameters
min_boundary = [1.2]             #minimum boundary
max_boundary = [10.0]            #maximum boundary
trust_region = 0.1               #maximum % move distance from best params
```

(continues on next page)

(continued from previous page)

```
first_params = [5.3]            #first parameters to try
```

random_complete_config.txt which contains a comprehensive list of options.

```
#Random Complete Options
#------------------------

#General options
max_num_runs = 20              #number of planned runs

#Random controller options
controller_type = 'random'
num_params = 2                      #number of parameters
min_boundary = [1.2,-2]             #minimum boundary
max_boundary = [10.0,4]            #maximum boundary
trust_region = [0.2,0.5]            #maximum move distance from best params
first_params = [5.1,-1.0]           #first parameters to try
```

2.6.3 Logging

You can control the filename of the logs and also the level which is reported to the file and the console. For more information see Link logging levels. The logging options are described in *logging_config.txt*.

```
#Logging Options
#---------------

log_filename = 'cl_run'              #Prefix for logging filename
file_log_level=logging.DEBUG        #Logging level saved in file
console_log_level=logging.WARNING   #Logging level presented to console, normally INFO
```

2.6.4 Extras

Extras refers to options related to post processing your data once the optimization is complete. Currently the only extra option is for visualizations. The extra options are described in *extras_config.txt*.

```
#Extra Options
#-------------

visualizations=False                        #whether plots should be presented after run
```

2.7 Contributing

If you use M-LOOP please consider contributing to the project. There are many quick and easy ways to help out.

- If you use M-LOOP be sure to cite the paper where it first used: 'Fast machine-learning online optimization of ultra-cold-atom experiments', Sci Rep 6, 25890 (2016).
- Star and watch the M-LOOP GitHub.

- Make a suggestion on what features you would like added, or report an issue, on the GitHub or by email.
- Contribute your own code to the M-LOOP GitHub, this could be the interface you designed, more options or a completely new solver.

Finally spread the word! Let others know the success you have had with M-LOOP and recommend they try it too.

2.7.1 Contributors

M-LOOP is written and maintained by Michael R Hush <MichaelRHush@gmail.com>

Other contributors, listed alphabetically, are:

- John W. Bastian - design, first demonstration
- Patrick J. Everitt - testing, design, first demonstration
- Kyle S. Hardman - design, first demonstration
- Anton van den Hengel - design, first demonstration
- Joe J. Hope - design, first demonstration
- Carlos C. N. Kuhn - first demonstration
- Andre N. Luiten - first demonstration
- Gordon D. McDonald - first demonstration
- Manju Perumbil - first demonstration
- Ian R. Petersen - first demonstration
- Ciaron D. Quinlivan - first demonstration
- Alex Ratcliff - testing
- Nick P. Robins - first demonstration
- Mahasen A. Sooriyabandara - first demonstration
- Richard Taylor - testing
- Paul B. Wigley - testing, design, first demonstration

2.8 M-LOOP API

M-LOOP can also be used as a library in python. This is particularly useful if the experiment you are optimizing can be controlled by python.

2.8.1 mloop

M-LOOP: Machine-Learning Online Optimization Packaage

Python package for performing automated, online optimization of scientific experiments or anything that can be computer controlled. The package employs machine learning algorithms to rapidly find optimal parameters for systems under control.

If you use this package please cite the article http://www.nature.com/articles/srep25890.

To contribute to the project or report a bug visit the project's github https://github.com/michaelhush/M-LOOP.

2.8.2 controllers

Module of all the controllers used in M-LOOP. The controllers, as the name suggests, control the interface to the experiment and all the learners employed to find optimal parameters.

class `mloop.controllers.`**`Controller`**(*interface*, *max_num_runs=inf*, *target_cost=-inf*, *max_num_runs_without_better_params=inf*, *controller_archive_filename='controller_archive'*, *controller_archive_file_type='txt'*, *archive_extra_dict=None*, *start_datetime=None*, ***kwargs*)

Bases: `object`

Abstract class for controllers. The controller controls the entire M-LOOP process. The controller for each algorithm all inherit from this class. The class stores a variety of data which all algorithms use and also all of the achiving and saving features. In order to implement your own controller class the minimum requirement is to add a learner to the learner variable. And implement the next_parameters method, where you provide the appropriate information to the learner and get the next parameters. See the RandomController for a simple implementation of a controller. Note the first three keywords are all possible halting conditions for the controller. If any of them are satisfied the controller will halt (meaning an and condition is used). Also creates an empty variable learner. The simplest way to make a working controller is to assign a learner of some kind to this variable, and add appropriate queues and events from it. :param interface: The interface process. Is run by learner. :type interface: interface

Keyword Arguments

- **`max_num_runs`** (*`Optional [float]`*) – The number of runs before the controller stops. If set to float('+inf') the controller will run forever. Default float('inf'), meaning the controller will run until another condition is met.
- **`target_cost`** (*`Optional [float]`*) – The target cost for the run. If a run achieves a cost lower than the target, the controller is stopped. Default float('-inf'), meaning the controller will run until another condition is met.
- **`max_num_runs_without_better_params`** (*`Otional [float]`*) – Puts a limit on the number of runs are allowed before a new better set of parameters is found. Default float('inf'), meaning the controller will run until another condition is met.
- **`controller_archive_filename`** (*`Optional [string]`*) – Filename for archive. Contains costs, parameter history and other details depending on the controller type. Default 'ControllerArchive.mat'
- **`controller_archive_file_type`** (*`Optional [string]`*) – File type for archive. Can be either 'txt' a human readable text file, 'pkl' a python dill file, 'mat' a matlab file or None if there is no archive. Default 'mat'.
- **`archive_extra_dict`** (*`Optional [dict]`*) – A dictionary with any extra variables that are to be saved to the archive. If None, nothing is added. Default None.
- **`start_datetime`** (*`Optional datetime`*) – Datetime for when controller was started.

`params_out_queue`

Queue for parameters to next be run by experiment.

Type queue

`costs_in_queue`

Queue for costs (and other details) that have been returned by experiment.

Type queue

end_interface
Event used to trigger the end of the interface

Type event

learner
The placeholder for the learner, creating this variable is the minimum requirement to make a working controller class.

Type None

learner_params_queue
The parameters queue for the learner

Type queue

learner_costs_queue
The costs queue for the learner

Type queue

end_learner
Event used to trigger the end of the learner

Type event

num_in_costs
Counter for the number of costs received.

Type int

num_out_params
Counter for the number of parameters received.

Type int

out_params
List of all parameters sent out by controller.

Type list

out_extras
Any extras associated with the output parameters.

Type list

in_costs
List of costs received by controller.

Type list

in_uncers
List of uncertainties receieved by controller.

Type list

best_cost
The lowest, and best, cost received by the learner.

Type float

best_uncer
The uncertainty associated with the best cost.

Type float

`best_params`
The best parameters recieved by the learner.

Type array

`best_index`
The run number that produced the best cost.

Type float

`_first_params``()`
Checks queue to get first parameters. :returns: Parameters for first experiment

`_get_cost_and_in_dict``()`
Get cost, uncertainty, parameters, bad and extra data from experiment. Stores in a list of history and also puts variables in their appropriate 'current' variables Note returns nothing, stores everything in the internal storage arrays and the curr_variables

`_next_params``()`
Abstract method. When implemented should send appropriate information to learner and get next parameters. :returns: Parameters for next experiment.

`_optimization_routine``()`
Runs controller main loop. Gives parameters to experiment and saves costs returned.

`_put_params_and_out_dict`(*params*, *param_type=None*, ***kwargs*)
Send parameters to queue and whatever additional keywords. Saves sent variables in appropriate storage arrays. :param params: array of values to be sent to file :type params: array

Keyword Arguments **`**kwargs`** – any additional data to be attached to file sent to experiment

`_shut_down``()`
Shutdown and clean up resources of the controller. end the learners, queue_listener and make one last save of archive.

`_start_up``()`
Start the learner and interface threads/processes.

`_update_controller_with_learner_attributes``()`
Update the controller with properties from the learner.

`check_end_conditions``()`
Check whether either of the three end contions have been met: number_of_runs, target_cost or max_num_runs_without_better_params. :returns: True, if the controlled should continue, False if the controller should end. :rtype: bool

`optimize``()`
Optimize the experiment. This code learner and interface processes/threads are launched and appropriately ended. Starts both threads and catches kill signals and shuts down appropriately.

`print_results``()`
Print results from optimization run to the logs

`save_archive``()`
Save the archive associated with the controller class. Only occurs if the filename for the archive is not None. Saves with the format previously set.

exception `mloop.controllers.`**`ControllerInterrupt`**
Bases: `Exception`

Exception that is raised when the controlled is ended with the end flag or event.

class mloop.controllers.**DifferentialEvolutionController**(*interface*, ***kwargs*)

Bases: *mloop.controllers.Controller*

Controller for the differential evolution learner. :param params_out_queue: Queue for parameters to next be run by experiment. :type params_out_queue: queue :param costs_in_queue: Queue for costs (and other details) that have been returned by experiment. :type costs_in_queue: queue :param **kwargs: Dictionary of options to be passed to Controller parent class and differential evolution learner. :type **kwargs: Optional [dict]

_next_params()

Gets next parameters from differential evolution learner.

class mloop.controllers.**GaussianProcessController**(*interface*, *num_params=None*, *min_boundary=None*, *max_boundary=None*, *trust_region=None*, *learner_archive_filename='learner_archive'*, *learner_archive_file_type='txt'*, ***kwargs*)

Bases: *mloop.controllers.MachineLearnerController*

Controller for the Gaussian Process solver. Primarily suggests new points from the Gaussian Process learner. However, during the initial few runs it must rely on a different optimization algorithm to get some points to seed the learner. :param interface: The interface to the experiment under optimization. :type interface: Interface :param **kwargs: Dictionary of options to be passed to MachineLearnerController parent class and Gaussian Process learner. :type **kwargs: Optional [dict]

Keyword Args:

class mloop.controllers.**MachineLearnerController**(*interface*, *training_type='differential_evolution'*, *machine_learner_type='machine_learner'*, *num_training_runs=None*, *no_delay=True*, *num_params=None*, *min_boundary=None*, *max_boundary=None*, *trust_region=None*, *learner_archive_filename='learner_archive'*, *learner_archive_file_type='txt'*, ***kwargs*)

Bases: *mloop.controllers.Controller*

Abstract Controller class for the machine learning based solvers. :param interface: The interface to the experiment under optimization. :type interface: Interface :param **kwargs: Dictionary of options to be passed to Controller parent class and initial training learner. :type **kwargs: Optional [dict]

Keyword Arguments

- **training_type** (*Optional [string]*) – The type for the initial training source can be 'random' for the random learner, 'nelder_mead' for the Nelder-Mead learner or 'differential_evolution' for the Differential Evolution learner. This learner is also called if the machine learning learner is too slow and a new point is needed. Default 'differential_evolution'.
- **num_training_runs** (*Optional [int]*) – The number of training runs to before starting the learner. If None, will be ten or double the number of parameters, whatever is larger.
- **no_delay** (*Optional [bool]*) – If True, there is never any delay between a returned cost and the next parameters to run for the experiment. In practice, this means if the machine learning learner has not prepared the next parameters in time the learner defined by the initial

training source is used instead. If false, the controller will wait for the machine learning learner to predict the next parameters and there may be a delay between runs.

_get_cost_and_in_dict()

Call _get_cost_and_in_dict() of parent Controller class. But also sends cost to machine learning learner and saves the cost if the parameters came from a trainer.

_next_params()

Gets next parameters from training learner.

_optimization_routine()

Overrides _optimization_routine. Uses the parent routine for the training runs. Implements a customized _optimization_routine when running the machine learning learner.

_put_params_and_out_dict(*params*)

Override _put_params_and_out_dict function, used when the training learner creates parameters. Makes the defualt param_type the training type and sets last_training_run_flag.

_shut_down()

Shutdown and clean up resources of the machine learning controller.

_start_up()

Runs pararent method and also starts training_learner.

print_results()

Adds some additional output to the results specific to controller.

class mloop.controllers.**NelderMeadController**(*interface*, ***kwargs*)

Bases: `mloop.controllers.Controller`

Controller for the Nelder-Mead solver. Suggests new parameters based on the Nelder-Mead algorithm. Can take no boundaries or hard boundaries. More details for the Nelder-Mead options are in the learners section. :param params_out_queue: Queue for parameters to next be run by experiment. :type params_out_queue: queue :param costs_in_queue: Queue for costs (and other details) that have been returned by experiment. :type costs_in_queue: queue :param **kwargs: Dictionary of options to be passed to Controller parent class and Nelder-Mead learner. :type **kwargs: Optional [dict]

_next_params()

Gets next parameters from Nelder-Mead learner.

class mloop.controllers.**NeuralNetController**(*interface*, *num_params=None*, *min_boundary=None*, *max_boundary=None*, *trust_region=None*, *learner_archive_filename='learner_archive'*, *learner_archive_file_type='txt'*, ***kwargs*)

Bases: `mloop.controllers.MachineLearnerController`

Controller for the Neural Net solver. Primarily suggests new points from the Neural Net learner. However, during the initial few runs it must rely on a different optimization algorithm to get some points to seed the learner. :param interface: The interface to the experiment under optimization. :type interface: Interface :param **kwargs: Dictionary of options to be passed to MachineLearnerController parent class and Neural Net learner. :type **kwargs: Optional [dict]

Keyword Args:

class mloop.controllers.**RandomController**(*interface*, ***kwargs*)

Bases: `mloop.controllers.Controller`

Controller that simply returns random variables for the next parameters. Costs are stored but do not influence future points picked. :param params_out_queue: Queue for parameters to next be run by experiment. :type params_out_queue: queue :param costs_in_queue: Queue for costs (and other details) that have been returned by experiment. :type costs_in_queue: queue :param **kwargs: Dictionary of options to be passed to Controller and Random Learner. :type **kwargs: Optional [dict]

`_next_params``()`

Sends cost uncer and bad tuple to learner then gets next parameters. :returns: Parameters for next experiment.

`mloop.controllers.`**`create_controller`**`(`*interface*, *controller_type='gaussian_process'*, ***controller_config_dict*`)`

Start the controller with the options provided. :param interface: Interface with queues and events to be passed to controller :type interface: interface

Keyword Arguments

- **`controller_type`** (*`Optional [str]`*) – Defines the type of controller can be 'random', 'nelder', 'gaussian_process' or 'neural_net'. Defaults to 'gaussian_process'.
- **`**controller_config_dict`** – Options to be passed to controller.

Returns threadible object which must be started with start() to get the controller running.

Return type *Controller*

Raises `ValueError` – if controller_type is an unrecognized string

2.8.3 interfaces

Module of the interfaces used to connect the controller to the experiment.

`class` `mloop.interfaces.`**`FileInterface`**`(`*interface_out_filename='exp_input'*, *interface_in_filename='exp_output'*, *interface_file_type='txt'*, ***kwargs*`)`

Bases: *`mloop.interfaces.Interface`*

Interfaces between the files produced by the experiment and the queues accessed by the controllers.

Parameters

- **`params_out_queue`** (*`queue`*) – Queue for parameters to next be run by experiment.
- **`costs_in_queue`** (*`queue`*) – Queue for costs (and other details) that have been returned by experiment.

Keyword Arguments

- **`interface_out_filename`** (*`Optional [string]`*) – filename for file written with parameters.
- **`interface_in_filename`** (*`Optional [string]`*) – filename for file written with parameters.
- **`interface_file_type`** (*`Optional [string]`*) – file type to be written either 'mat' for matlab or 'txt' for readible text file. Defaults to 'txt'.

`get_next_cost_dict``(`*params_dict*`)`

Implementation of file read in and out. Put parameters into a file and wait for a cost file to be returned.

`class` `mloop.interfaces.`**`Interface`**`(`*interface_wait=1*, ***kwargs*`)`

Bases: `threading.Thread`

A abstract class for interfaces which populate the costs_in_queue and read from the params_out_queue. Inherits from Thread

Parameters

- **interface_wait** (*Optional [float]*) – Time between polling when needed in interface.
- **params_out_queue** (*queue*) – Queue for parameters to next be run by experiment.
- **costs_in_queue** (*queue*) – Queue for costs (and other details) that have been returned by experiment.
- **end_event** (*event*) – Event which triggers the end of the interface.

Keyword Arguments **interface_wait** (*float*) – Wait time when polling for files or queues is needed.

get_next_cost_dict (*params_dict*)

Abstract method. This is the only method that needs to be implemented to make a working interface. Given the parameters the interface must then produce a new cost. This may occur by running an experiment or program. If you wish to abruptly end this interface for whatever rease please raise the exception InterfaceInterrupt, which will then be safely caught.

Parameters **params_dict** (*dictionary*) – A dictionary containing the parameters. Use params_dict['params'] to access them.

Returns The cost and other properties derived from the experiment when it was run with the parameters. If just a cost was produced provide {'cost':[float]}, if you also have an uncertainty provide {'cost':[float],'uncer':[float]}. If the run was bad you can simply provide {'bad':True}. For completeness you can always provide all three using {'cost':[float],'uncer':[float],'bad':[bool]}. Providing any extra keys will also be saved byt he controller.

Return type cost_dict (dictionary)

run()

The run sequence for the interface. This method does not need to be overloaded create a working interface.

exception mloop.interfaces.**InterfaceInterrupt**

Bases: Exception

Exception that is raised when the interface is ended with the end event, or some other interruption.

class mloop.interfaces.**ShellInterface**(*command='./run_exp'*, *params_args_type='direct'*, ***kwargs*)

Bases: *mloop.interfaces.Interface*

Interface for running programs from the shell.

Parameters

- **params_out_queue** (*queue*) – Queue for parameters to next be run by experiment.
- **costs_in_queue** (*queue*) – Queue for costs (and other details) that have been returned by experiment.

Keyword Arguments

- **command** (*Optional [string]*) – The command used to run the experiment. Default './run_exp'
- **params_args_type** (*Optional [string]*) – The style used to pass parameters. Can be 'direct' or 'named'. If 'direct' it is assumed the parameters are fed directly to the program. For example if I wanted to run the parameters [7,5,9] with the command './run_exp' I would use the syntax:

```
./run_exp 7 5 9
```

'named' on the other hand requires an option for each parameter. The options should be name –param1, –param2 etc. The same example as before would be

```
./run_exp --param1 7 --param2 5 --param3 9
```

Default 'direct'.

get_next_cost_dict (*params_dict*)
Implementation of running a command with parameters on the command line and reading the result.

class mloop.interfaces.**TestInterface** (*test_landscape=None*, ***kwargs*)
Bases: *mloop.interfaces.Interface*

Interface for testing. Returns fake landscape data directly to learner.

Parameters

- **params_out_queue** (*queue*) – Parameters to be used to evaluate fake landscape.
- **costs_in_queue** (*queue*) – Queue for costs (and other details) that have been calculated from fake landscape.

Keyword Arguments

- **test_landscape** (*Optional [TestLandscape]*) – Landscape that can be given a set of parameters and a cost and other values. If None creates a the default landscape. Default None
- **out_queue_wait** (*Optional [float]*) – Time in seconds to wait for queue before checking end flag.

get_next_cost_dict (*params_dict*)
Test implementation. Gets the next cost from the test_landscape.

mloop.interfaces.**create_interface** (*interface_type='file'*, ***interface_config_dict*)
Start a new interface with the options provided.

Parameters

- **interface_type** (*Optional [str]*) – Defines the type of interface, can be 'file', 'shell' or 'test'. Default 'file'.
- ****interface_config_dict** – Options to be passed to interface.

Returns An interface as defined by the keywords

Return type interface

2.8.4 launchers

Modules of launchers used to start M-LOOP.

mloop.launchers.**_pop_extras_kwargs** (*kwargs*)
Remove the keywords used in the extras section (if present), and return them.

Returns tuple made of (extras_kwargs, kwargs), where extras_kwargs are keywords for the extras and kwargs are the others that were provided.

mloop.launchers.**launch_extras** (*controller*, *visualizations=True*, ***kwargs*)
Launch post optimization extras. Including visualizations.

Keyword Arguments **visualizations** (*Optional [bool]*) – If true run default visualizations for the controller. Default false.

`mloop.launchers.`**`launch_from_file`**(*config_filename*, ***kwargs*)

Launch M-LOOP using a configuration file. See configuration file documentation.

Parameters

- **`config_filename`** (`str`) – Filename of configuration file
- **`**kwargs`** – keywords that override the keywords in the file.

Returns Controller for optimization.

Return type controller (*Controller*)

2.8.5 learners

Module of learners used to determine what parameters to try next given previous cost evaluations.

Each learner is created and controlled by a controller.

`class` `mloop.learners.`**`DifferentialEvolutionLearner`**(*first_params=None*, *trust_region=None*, *evolution_strategy='best1'*, *population_size=15*, *mutation_scale=(0.5, 1)*, *cross_over_probability=0.7*, *restart_tolerance=0.01*, ***kwargs*)

Bases: `mloop.learners.Learner`, `threading.Thread`

Adaption of the differential evolution algorithm in scipy.

Parameters

- **`params_out_queue`** (`queue`) – Queue for parameters sent to controller.
- **`costs_in_queue`** (`queue`) – Queue for costs for gaussian process. This must be tuple
- **`end_event`** (`event`) – Event to trigger end of learner.

Keyword Arguments

- **`first_params`** (`Optional [array]`) – The first parameters to test. If None will just randomly sample the initial condition. Default None.
- **`trust_region`** (`Optional [float or array]`) – The trust region defines the maximum distance the learner will travel from the current best set of parameters. If None, the learner will search everywhere. If a float, this number must be between 0 and 1 and defines maximum distance the learner will venture as a percentage of the boundaries. If it is an array, it must have the same size as the number of parameters and the numbers define the maximum absolute distance that can be moved along each direction.
- **`evolution_strategy`** (`Optional [string]`) – the differential evolution strategy to use, options are 'best1', 'best1', 'rand1' and 'rand2'. The default is 'best2'.
- **`population_size`** (`Optional [int]`) – multiplier proportional to the number of parameters in a generation. The generation population is set to population_size * parameter_num. Default 15.
- **`mutation_scale`** (`Optional [tuple]`) – The mutation scale when picking new points. Otherwise known as differential weight. When provided as a tuple (min,max) a mutation constant is picked randomly in the interval. Default (0.5,1.0).
- **`cross_over_probability`** (`Optional [float]`) – The recombination constand or crossover probability, the probability a new points will be added to the population.

- **restart_tolerance** (*Optional [float]*) – when the current population have a spread less than the initial tolerance, namely stdev(curr_pop) < restart_tolerance stdev(init_pop), it is likely the population is now in a minima, and so the search is started again.

has_trust_region
Whether the learner has a trust region.

Type bool

num_population_members
The number of parameters in a generation.

Type int

params_generations
History of the parameters generations. A list of all the parameters in the population, for each generation created.

Type list

costs_generations
History of the costs generations. A list of all the costs in the population, for each generation created.

Type list

init_std
The initial standard deviation in costs of the population. Calucalted after sampling (or resampling) the initial population.

Type float

curr_std
The current standard devation in costs of the population. Calculated after sampling each generation.

Type float

_best1(*index*)
Use best parameters and two others to generate mutation.

Parameters **index** (*int*) – Index of member to mutate.

_best2(*index*)
Use best parameters and four others to generate mutation.

Parameters **index** (*int*) – Index of member to mutate.

_rand1(*index*)
Use three random parameters to generate mutation.

Parameters **index** (*int*) – Index of member to mutate.

_rand2(*index*)
Use five random parameters to generate mutation.

Parameters **index** (*int*) – Index of member to mutate.

generate_population()
Sample a new random set of variables

mutate(*index*)
Mutate the parameters at index.

Parameters **index** (*int*) – Index of the point to be mutated.

next_generation()
Evolve the population by a single generation

random_index_sample(*index*, *num_picks*)
Randomly select a num_picks of indexes, without index.

Parameters

- **index** (*int*) – The index that is not included
- **num_picks** (*int*) – The number of picks.

run()
Runs the Differential Evolution Learner.

save_generation()
Save history of generations.

update_archive()
Update the archive.

class mloop.learners.**GaussianProcessLearner**(*length_scale=None*, *update_hyperparameters=True*, *cost_has_noise=True*, *noise_level=1.0*, *trust_region=None*, *default_bad_cost=None*, *default_bad_uncertainty=None*, *minimum_uncertainty=1e-08*, *gp_training_filename=None*, *gp_training_file_type='txt'*, *predict_global_minima_at_end=True*, ***kwargs*)
Bases: *mloop.learners.Learner*, multiprocessing.context.Process

Gaussian process learner. Generats new parameters based on a gaussian process fitted to all previous data.

Parameters

- **params_out_queue** (*queue*) – Queue for parameters sent to controller.
- **costs_in_queue** (*queue*) – Queue for costs for gaussian process. This must be tuple
- **end_event** (*event*) – Event to trigger end of learner.

Keyword Arguments

- **length_scale** (*Optional [array]*) – The initial guess for length scale(s) of the gaussian process. The array can either of size one or the number of parameters or None. If it is size one, it is assumed all the correlation lengths are the same. If it is the number of the parameters then all the parameters have their own independent length scale. If it is None, it is assumed all the length scales should be independent and they are all given an initial value of 1. Default None.
- **cost_has_noise** (*Optional [bool]*) – If true the learner assumes there is common additive white noise that corrupts the costs provided. This noise is assumed to be on top of the uncertainty in the costs (if it is provided). If false, it is assumed that there is no noise in the cost (or if uncertainties are provided no extra noise beyond the uncertainty). Default True.
- **noise_level** (*Optional [float]*) – The initial guess for the noise level in the costs, is only used if cost_has_noise is true. Default 1.0.
- **update_hyperparameters** (*Optional [bool]*) – Whether the length scales and noise estimate should be updated when new data is provided. Is set to true by default.

- **trust_region** (*Optional [float or array]*) – The trust region defines the maximum distance the learner will travel from the current best set of parameters. If None, the learner will search everywhere. If a float, this number must be between 0 and 1 and defines maximum distance the learner will venture as a percentage of the boundaries. If it is an array, it must have the same size as the number of parameters and the numbers define the maximum absolute distance that can be moved along each direction.
- **default_bad_cost** (*Optional [float]*) – If a run is reported as bad and default_bad_cost is provided, the cost for the bad run is set to this default value. If default_bad_cost is None, then the worst cost received is set to all the bad runs. Default None.
- **default_bad_uncertainty** (*Optional [float]*) – If a run is reported as bad and default_bad_uncertainty is provided, the uncertainty for the bad run is set to this default value. If default_bad_uncertainty is None, then the uncertainty is set to a tenth of the best to worst cost range. Default None.
- **minimum_uncertainty** (*Optional [float]*) – The minimum uncertainty associated with provided costs. Must be above zero to avoid fitting errors. Default 1e-8.
- **predict_global_minima_at_end** (*Optional [bool]*) – If True finds the global minima when the learner is ended. Does not if False. Default True.

all_params
Array containing all parameters sent to learner.

Type array

all_costs
Array containing all costs sent to learner.

Type array

all_uncers
Array containing all uncertainties sent to learner.

Type array

scaled_costs
Array contaning all the costs scaled to have zero mean and a standard deviation of 1. Needed for training the gaussian process.

Type array

bad_run_indexs
list of indexes to all runs that were marked as bad.

Type list

best_cost
Minimum received cost, updated during execution.

Type float

best_params
Parameters of best run. (reference to element in params array).

Type array

best_index
index of the best cost and params.

Type int

worst_cost
Maximum received cost, updated during execution.

Type float

worst_index
index to run with worst cost.

Type int

cost_range
Difference between worst_cost and best_cost

Type float

generation_num
Number of sets of parameters to generate each generation. Set to 5.

Type int

length_scale_history
List of length scales found after each fit.

Type list

noise_level_history
List of noise levels found after each fit.

Type list

fit_count
Counter for the number of times the gaussian process has been fit.

Type int

cost_count
Counter for the number of costs, parameters and uncertainties added to learner.

Type int

params_count
Counter for the number of parameters asked to be evaluated by the learner.

Type int

gaussian_process
Gaussian process that is fitted to data and used to make predictions

Type GaussianProcessRegressor

cost_scaler
Scaler used to normalize the provided costs.

Type StandardScaler

has_trust_region
Whether the learner has a trust region.

Type bool

create_gaussian_process()
Create the initial Gaussian process.

find_global_minima()
Performs a quick search for the predicted global minima from the learner. Does not return any values, but creates the following attributes.

predicted_best_parameters
the parameters for the predicted global minima
Type array

predicted_best_cost
the cost at the predicted global minima
Type float

predicted_best_uncertainty
the uncertainty of the predicted global minima
Type float

find_next_parameters()
Returns next parameters to find. Increments counters and bias function appropriately.

Returns Returns next parameters from biased cost search.

Return type next_params (array)

fit_gaussian_process()
Fit the Gaussian process to the current data

get_params_and_costs()
Get the parameters and costs from the queue and place in their appropriate all_[type] arrays. Also updates bad costs, best parameters, and search boundaries given trust region.

predict_biased_cost(*params*)

Predicts the biased cost at the given parameters. The bias function is: biased_cost = cost_bias*pred_cost - uncer_bias*pred_uncer

Returns Biased cost predicted at the given parameters

Return type pred_bias_cost (float)

predict_cost(*params*)
Produces a prediction of cost from the gaussian process at params.

Returns Predicted cost at paramters

Return type float

run()
Starts running the Gaussian process learner. When the new parameters event is triggered, reads the cost information provided and updates the Gaussian process with the information. Then searches the Gaussian process for new optimal parameters to test based on the biased cost. Parameters to test next are put on the output parameters queue.

update_archive()
Update the archive.

update_bads()
Best and/or worst costs have changed, update the values associated with bad runs accordingly.

update_bias_function()
Set the constants for the cost bias function.

update_search_params()
Update the list of parameters to use for the next search.

update_search_region()
If trust boundaries is not none, updates the search boundaries based on the defined trust region.

wait_for_new_params_event()

Waits for a new parameters event and starts a new parameter generation cycle. Also checks end event and will break if it is triggered.

class mloop.learners.**Learner**(*num_params=None*, *min_boundary=None*, *max_boundary=None*, *learner_archive_filename='learner_archive'*, *learner_archive_file_type='txt'*, *start_datetime=None*, ***kwargs*)

Bases: `object`

Base class for all learners. Contains default boundaries and some useful functions that all learners use.

The class that inherits from this class should also inherit from threading.Thread or multiprocessing.Process, depending if you need the learner to be a genuine parallel process or not.

Keyword Arguments

- **num_params** (*Optional [int]*) – The number of parameters to be optimized. If None defaults to 1. Default None.
- **min_boundary** (*Optional [array]*) – Array with minimimum values allowed for each parameter. Note if certain values have no minimum value you can set them to -inf for example [-1, 2, float('-inf')] is a valid min_boundary. If None sets all the boundaries to '-1'. Default None.
- **max_boundary** (*Optional [array]*) – Array with maximum values allowed for each parameter. Note if certain values have no maximum value you can set them to +inf for example [0, float('inf'),3,-12] is a valid max_boundary. If None sets all the boundaries to '1'. Default None.
- **learner_archive_filename** (*Optional [string]*) – Name for python archive of the learners current state. If None, no archive is saved. Default None. But this is typically overloaded by the child class.
- **learner_archive_file_type** (*Optional [string]*) – File type for archive. Can be either 'txt' a human readable text file, 'pkl' a python dill file, 'mat' a matlab file or None if there is no archive. Default 'mat'.
- **log_level** (*Optional [int]*) – Level for the learners logger. If None, set to warning. Default None.
- **start_datetime** (*Optional [datetime]*) – Start date time, if None, is automatically generated.

params_out_queue

Queue for parameters created by learner.

Type queue

costs_in_queue

Queue for costs to be used by learner.

Type queue

end_event

Event to trigger end of learner.

Type event

_set_trust_region(*trust_region*)

Sets trust region properties for learner that have this. Common function for learners with trust regions.

Parameters **trust_region** (*float or array*) – Property defines the trust region.

_shut_down()
Shut down and perform one final save of learner.

check_in_boundary(*param*)
Check give parameters are within stored boundaries

Parameters **param** (*array*) – array of parameters

Returns True if the parameters are within boundaries, False otherwise.

Return type bool

check_in_diff_boundary(*param*)
Check given distances are less than the boundaries

Parameters **param** (*array*) – array of distances

Returns True if the distances are smaller or equal to boundaries, False otherwise.

Return type bool

check_num_params(*param*)
Check the number of parameters is right.

put_params_and_get_cost(*params*, ***kwargs*)
Send parameters to queue and whatever additional keywords. Saves sent variables in appropriate storage arrays.

Parameters **params** (*array*) – array of values to be sent to file

Returns cost from the cost queue

save_archive()
Save the archive associated with the learner class. Only occurs if the filename for the archive is not None. Saves with the format previously set.

update_archive()
Abstract method for update to the archive. To be implemented by child class.

exception mloop.learners.**LearnerInterrupt**
Bases: Exception

Exception that is raised when the learner is ended with the end flag or event.

class mloop.learners.**NelderMeadLearner**(*initial_simplex_corner=None*, *initial_simplex_displacements=None*, *initial_simplex_scale=None*, ***kwargs*)
Bases: *mloop.learners.Learner*, threading.Thread

Nelder-Mead learner. Executes the Nelder-Mead learner algorithm and stores the needed simplex to estimate the next points.

Parameters

- **params_out_queue** (*queue*) – Queue for parameters from controller.
- **costs_in_queue** (*queue*) – Queue for costs for nelder learner. The queue should be populated with cost (float) corresponding to the last parameter sent from the Nelder-Mead Learner. Can be a float('inf') if it was a bad run.
- **end_event** (*event*) – Event to trigger end of learner.

Keyword Arguments

- **initial_simplex_corner** (*Optional [array]*) – Array for the initial set of parameters, which is the lowest corner of the initial simplex. If None the initial parameters are randomly sampled if the boundary conditions are provided, or all are set to 0 if boundary conditions are not provided.

- **`initial_simplex_displacements`** (*`Optional [array]`*) – Array used to construct the initial simplex. Each array is the positive displacement of the parameters above the init_params. If None and there are no boundary conditions, all are set to 1. If None and there are boundary conditions assumes the initial conditions are scaled. Default None.
- **`initial_simplex_scale`** (*`Optional [float]`*) – Creates a simplex using a the boundary conditions and the scaling factor provided. If None uses the init_simplex if provided. If None and init_simplex is not provided, but boundary conditions are is set to 0.5. Default None.

`init_simplex_corner`
Parameters for the corner of the initial simple used.

Type array

`init_simplex_disp`
Parameters for the displacements about the simplex corner used to create the initial simple.

Type array

`simplex_params`
Parameters of the current simplex

Type array

`simplex_costs`
Costs associated with the parameters of the current simplex

Type array

`run``()`
Runs Nelder-Mead algorithm to produce new parameters given costs, until end signal is given.

`update_archive``()`
Update the archive.

`class` `mloop.learners.`**`NeuralNetLearner`**(*trust_region=None*, *default_bad_cost=None*, *default_bad_uncertainty=None*, *nn_training_filename=None*, *nn_training_file_type='txt'*, *minimum_uncertainty=1e-08*, *predict_global_minima_at_end=True*, ***kwargs*)
Bases: *`mloop.learners.Learner`*, `multiprocessing.context.Process`

Learner that uses a neural network for function approximation.

Parameters

- **`params_out_queue`** (*`queue`*) – Queue for parameters sent to controller.
- **`costs_in_queue`** (*`queue`*) – Queue for costs.
- **`end_event`** (*`event`*) – Event to trigger end of learner.

Keyword Arguments

- **`trust_region`** (*`Optional [float or array]`*) – The trust region defines the maximum distance the learner will travel from the current best set of parameters. If None, the learner will search everywhere. If a float, this number must be between 0 and 1 and defines maximum distance the learner will venture as a percentage of the boundaries. If it is an array, it must have the same size as the number of parameters and the numbers define the maximum absolute distance that can be moved along each direction.

- **`default_bad_cost`** (*`Optional [float]`*) – If a run is reported as bad and default_bad_cost is provided, the cost for the bad run is set to this default value. If default_bad_cost is None, then the worst cost received is set to all the bad runs. Default None.
- **`default_bad_uncertainty`** (*`Optional [float]`*) – If a run is reported as bad and default_bad_uncertainty is provided, the uncertainty for the bad run is set to this default value. If default_bad_uncertainty is None, then the uncertainty is set to a tenth of the best to worst cost range. Default None.
- **`minimum_uncertainty`** (*`Optional [float]`*) – The minimum uncertainty associated with provided costs. Must be above zero to avoid fitting errors. Default 1e-8.
- **`predict_global_minima_at_end`** (*`Optional [bool]`*) – If True finds the global minima when the learner is ended. Does not if False. Default True.

`all_params`
Array containing all parameters sent to learner.

Type array

`all_costs`
Array containing all costs sent to learner.

Type array

`all_uncers`
Array containing all uncertainties sent to learner.

Type array

`scaled_costs`
Array contaning all the costs scaled to have zero mean and a standard deviation of 1.

Type array

`bad_run_indexs`
list of indexes to all runs that were marked as bad.

Type list

`best_cost`
Minimum received cost, updated during execution.

Type float

`best_params`
Parameters of best run. (reference to element in params array).

Type array

`best_index`
index of the best cost and params.

Type int

`worst_cost`
Maximum received cost, updated during execution.

Type float

worst_index
index to run with worst cost.

Type int

cost_range
Difference between worst_cost and best_cost

Type float

generation_num
Number of sets of parameters to generate each generation. Set to 5.

Type int

noise_level_history
List of noise levels found after each fit.

Type list

cost_count
Counter for the number of costs, parameters and uncertainties added to learner.

Type int

params_count
Counter for the number of parameters asked to be evaluated by the learner.

Type int

neural_net
Neural net that is fitted to data and used to make predictions.

Type NeuralNet

cost_scaler
Scaler used to normalize the provided costs.

Type StandardScaler

cost_scaler_init_index
The number of params to use to initialise cost_scaler.

Type int

has_trust_region
Whether the learner has a trust region.

Type bool

_fit_neural_net(*index*)
Fits a neural net to the data.

cost_scaler must have been fitted before calling this method.

_init_cost_scaler()
Initialises the cost scaler. cost_scaler_init_index must be set.

create_neural_net()
Creates the neural net. Must be called from the same process as fit_neural_net, predict_cost and predict_costs_from_param_array.

find_global_minima()
Performs a quick search for the predicted global minima from the learner. Does not return any values, but creates the following attributes.

`predicted_best_parameters`
the parameters for the predicted global minima
Type array

`predicted_best_cost`
the cost at the predicted global minima
Type float

`find_next_parameters`(*net_index=None*)
Returns next parameters to find. Increments counters appropriately.

Returns Returns next parameters from cost search.

Return type next_params (array)

`get_losses`()

`get_params_and_costs`()
Get the parameters and costs from the queue and place in their appropriate all_[type] arrays. Also updates bad costs, best parameters, and search boundaries given trust region.

`import_neural_net`()
Imports neural net parameters from the training dictionary provided at construction. Must be called from the same process as fit_neural_net, predict_cost and predict_costs_from_param_array. You must call exactly one of this and create_neural_net before calling other methods.

`predict_cost`(*params*, *net_index=None*)
Produces a prediction of cost from the neural net at params.

Returns Predicted cost at paramters

Return type float

`predict_cost_gradient`(*params*, *net_index=None*)
Produces a prediction of the gradient of the cost function at params.

Returns Predicted gradient at paramters

Return type float

`predict_costs_from_param_array`(*params*, *net_index=None*)
Produces a prediction of costs from an array of params.

Returns Predicted cost at paramters

Return type float

`run`()
Starts running the neural network learner. When the new parameters event is triggered, reads the cost information provided and updates the neural network with the information. Then searches the neural network for new optimal parameters to test based on the biased cost. Parameters to test next are put on the output parameters queue.

`update_archive`()
Update the archive.

`update_bads`()
Best and/or worst costs have changed, update the values associated with bad runs accordingly.

`update_search_params`()
Update the list of parameters to use for the next search.

`update_search_region`()
If trust boundaries is not none, updates the search boundaries based on the defined trust region.

wait_for_new_params_event()

Waits for a new parameters event and starts a new parameter generation cycle. Also checks end event and will break if it is triggered.

class mloop.learners.**RandomLearner**(*trust_region=None*, *first_params=None*, ***kwargs*)

Bases: *mloop.learners.Learner*, threading.Thread

Random learner. Simply generates new parameters randomly with a uniform distribution over the boundaries. Learner is perhaps a misnomer for this class.

Parameters ****kwargs** (*Optional dict*) – Other values to be passed to Learner.

Keyword Arguments

- **min_boundary** (*Optional [array]*) – If set to None, overrides default learner values and sets it to a set of value 0. Default None.
- **max_boundary** (*Optional [array]*) – If set to None overides default learner values and sets it to an array of value 1. Default None.
- **first_params** (*Optional [array]*) – The first parameters to test. If None will just randomly sample the initial condition.
- **trust_region** (*Optional [float or array]*) – The trust region defines the maximum distance the learner will travel from the current best set of parameters. If None, the learner will search everywhere. If a float, this number must be between 0 and 1 and defines maximum distance the learner will venture as a percentage of the boundaries. If it is an array, it must have the same size as the number of parameters and the numbers define the maximum absolute distance that can be moved along each direction.

run()

Puts the next parameters on the queue which are randomly picked from a uniform distribution between the minimum and maximum boundaries when a cost is added to the cost queue.

2.8.6 testing

Module of classes used to test M-LOOP.

class mloop.testing.**FakeExperiment**(*test_landscape=None*, *experiment_file_type='txt'*, *exp_wait=0*, *poll_wait=1*, ***kwargs*)

Bases: threading.Thread

Pretends to be an experiment and reads files and prints files based on the costs provided by a TestLandscape. Executes as a thread.

Keyword Arguments

- **test_landscape** (*Optional TestLandscape*) – landscape to generate costs from.
- **experiment_file_type** (*Optional [string]*) – currently supports: 'txt' where the output is a text file with the parameters as a list of numbers, and 'mat' a matlab file with variable parameters with the next_parameters. Default is 'txt'.

Attributes self.end_event (Event): Used to trigger end of experiment.

run()

Implementation of file read in and out. Put parameters into a file and wait for a cost file to be returned.

set_landscape(*test_landscape*)

Set new test landscape.

Parameters **test_landscape** (`TestLandscape`) – Landscape to generate costs from.

class `mloop.testing.`**TestLandscape**(*num_params=1*)

Bases: `object`

Produces fake landscape data for testing, default functions are set for each of the methods which can then be over ridden.

Keyword Arguments **num_parameters** (*Optional [int]*) – Number of parameters for landscape, defaults to 1.

get_cost_dict(*params*)

Return cost from fake landscape given parameters.

Parameters **params** (*array*) – Parameters to evaluate cost.

set_bad_region(*min_boundary*, *max_boundary*, *bad_cost=None*, *bad_uncer=None*)

Adds a region to landscape that is reported as bad.

Parameters

- **min_boundary** (*array*) – mininum boundary for bad region
- **max_boundary** (*array*) – maximum boundary for bad region

set_default_landscape()

Set landscape functions to their defaults

set_noise_function(*proportional=0.0*, *absolute=0.0*)

Adds noise to the function.

with the formula:

```
c'(c,x) = c (1 + s_p p) + s_a a
```

where s_i are gaussian random variables, p is the proportional noise factor and a is the absolute noise factor, and c is the cost before noise is added

the uncertainty is then:

```
u = sqrt((cp)^2 + a^2)
```

Keyword Arguments

- **proportional** (*Optional [float]*) – the proportional factor. Defaults to 0
- **absolute** (*Optional [float]*) – the absolute factor. Defaults to 0

set_quadratic_landscape(*minimum=None*, *scale=None*)

Set deterministic part of landscape to be a quadratic.

with the formula:

```
c(x) = \sum_i a_i * (x_i - x_0,i)^2
```

where x_i are the parameters, x_0,i is the location of the minimum and a_i are the scaling factors.

Keyword Arguments

- **minimum** (*Optional [array]*) – Location of the minimum. If set to None is at the origin. Default None.
- **scales** (*Optional [array]*) – scaling of quadratic along the dimention specified. If set to None the scaling is one.

set_random_quadratic_landscape(*min_region*, *max_region*, *random_scale=True*, *min_scale=0*, *max_scale=3*)

Make a quadratic function with a minimum randomly placed in a region with random scales

Parameters

- **min_region** (*array*) – minimum region boundary
- **max_region** (*array*) – maximum region boundary

Keyword Arguments

- **random_scale** (*Optional bool*) – If true randomize the scales of the parameters. Default True.
- **min_scale** (*float*) – Natural log of minimum scale factor. Default 0.
- **max_scale** (*float*) – Natural log of maximum scale factor. Default 3.

2.8.7 utilities

Module of common utility methods and attributes used by all the modules.

class mloop.utilities.**NullQueueListener**

Bases: `object`

Shell class with start and stop functions that do nothing. Queue listener is not implemented in python 2. Current fix is to simply use the multiprocessing class to pipe straight to the cmd line if running on python 2. This is class is just a placeholder.

start()

Does nothing

stop()

Does nothing

mloop.utilities.**_config_logger**(*log_filename='M-LOOP_'*, *file_log_level=10*, *console_log_level=20*, ***kwargs*)

Configure and the root logger.

Keyword Arguments

- **log_filename** (*Optional [string]*) – Filename prefix for log. Default MLOOP run . If None, no file handler is created
- **file_log_level** (*Optional[int]*) – Level of log output for file, default is logging.DEBUG = 10
- **console_log_level** (*Optional[int]*) – Level of log output for console, defalut is logging.INFO = 20

Returns Dict with extra keywords not used by the logging configuration.

Return type dictionary

mloop.utilities.**check_file_type_supported**(*file_type*)

Checks whether the file type is supported

Returns True if file_type is supported, False otherwise.

Return type bool

mloop.utilities.**config_logger**(***kwargs*)

Wrapper for _config_logger.

mloop.utilities.**datetime_to_string**(*datetime*)
Method for changing a datetime into a standard string format used by all packages.

mloop.utilities.**dict_to_txt_file**(*tdict*, *filename*)
Method for writing a dict to a file with syntax similar to how files are input.

Parameters

- **tdict** (*dict*) – Dictionary to be written to file.
- **filename** (*string*) – Filename for file.

mloop.utilities.**get_dict_from_file**(*filename*, *file_type*)
Method for getting a dictionary from a file, of a given format.

Parameters

- **filename** – The filename for the file.
- **file_type** – The file_type for the file. Can be 'mat' for matlab, 'txt' for text, or 'pkl' for pickle.

Returns Dictionary of values in file.

Return type dict

mloop.utilities.**safe_cast_to_array**(*in_array*)
Attempts to safely cast the input to an array. Takes care of border cases

Parameters **in_array** (*array or equivalent*) – The array (or otherwise) to be converted to a list.

Returns array that has been squeezed and 0-D cases change to 1-D cases

Return type array

mloop.utilities.**safe_cast_to_list**(*in_array*)
Attempts to safely cast a numpy array to a list, if not a numpy array just casts to list on the object.

Parameters **in_array** (*array or equivalent*) – The array (or otherwise) to be converted to a list.

Returns List of elements from in_array

Return type list

mloop.utilities.**save_dict_to_file**(*dictionary*, *filename*, *file_type*)
Method for saving a dictionary to a file, of a given format.

Parameters

- **dictionary** – The dictionary to be saved in the file.
- **filename** – The filename for the saved file
- **file_type** – The file_type for the saved file. Can be 'mat' for matlab, 'txt' for text, or 'pkl' for pickle.

mloop.utilities.**txt_file_to_dict**(*filename*)
Method for taking a file and changing it to a dict. Every line in file is a new entry for the dictionary and each element should be written as:

```
[key] = [value]
```

White space does not matter.

Parameters **filename** (*string*) – Filename of file.

Returns Dictionary of values in file.

Return type dict

2.8.8 visualizations

Module of classes used to create visualizations of data produced by the experiment and learners.

class mloop.visualizations.**ControllerVisualizer**(*filename*, *file_type='pkl'*, ***kwargs*)

Bases: `object`

ControllerVisualizer creates figures from a Controller Archive.

Parameters **filename** (*String*) – Filename of the GaussianProcessLearner archive.

Keyword Arguments **file_type** (*String*) – Can be 'mat' for matlab, 'pkl' for pickle or 'txt' for text. Default 'pkl'.

plot_cost_vs_run()

Create a plot of the costs versus run number.

plot_parameters_vs_cost()

Create a plot of the parameters versus run number.

plot_parameters_vs_run()

Create a plot of the parameters versus run number.

class mloop.visualizations.**DifferentialEvolutionVisualizer**(*filename*, *file_type='pkl'*, ***kwargs*)

Bases: `object`

DifferentialEvolutionVisualizer creates figures from a differential evolution archive.

Parameters **filename** (*String*) – Filename of the DifferentialEvolutionVisualizer archive.

Keyword Arguments **file_type** (*String*) – Can be 'mat' for matlab, 'pkl' for pickle or 'txt' for text. Default 'pkl'.

plot_costs_vs_generations()

Create a plot of the costs versus run number.

plot_params_vs_generations()

Create a plot of the parameters versus run number.

class mloop.visualizations.**GaussianProcessVisualizer**(*filename*, *file_type='pkl'*, ***kwargs*)

Bases: *mloop.learners.GaussianProcessLearner*

GaussianProcessVisualizer extends of GaussianProcessLearner, designed not to be used as a learner, but to instead post process a GaussianProcessLearner archive file and produce useful data for visualization of the state of the learner. Fixes the Gaussian process hyperparameters to what was last found during the run.

Parameters **filename** (*String*) – Filename of the GaussianProcessLearner archive.

Keyword Arguments **file_type** (*String*) – Can be 'mat' for matlab, 'pkl' for pickle or 'txt' for text. Default 'pkl'.

plot_cross_sections()

Produce a figure of the cross section about best cost and parameters

plot_hyperparameters_vs_run()

return_cross_sections(*points=100*, *cross_section_center=None*)

Finds the predicted global minima, then returns a list of vectors of parameters values, costs and uncertainties, corresponding to the 1D cross sections along each parameter axis through the predicted global minima.

Keyword Arguments

- **points** (*int*) – the number of points to sample along each cross section. Default value is 100.
- **cross_section_center** (*array*) – parameter array where the centre of the cross section should be taken. If None, the parameters for the best returned cost are used.

Returns a tuple (cross_arrays, cost_arrays, uncer_arrays) cross_parameter_arrays (list): a list of arrays for each cross section, with the values of the varied parameter going from the minimum to maximum value. cost_arrays (list): a list of arrays for the costs evaluated along each cross section about the minimum. uncertainty_arrays (list): a list of uncertainties

run()

Overides the GaussianProcessLearner multiprocessor run routine. Does nothing but makes a warning.

class mloop.visualizations.**NeuralNetVisualizer**(*filename*, *file_type='pkl'*, ***kwargs*)

Bases: *mloop.learners.NeuralNetLearner*

NeuralNetVisualizer extends of NeuralNetLearner, designed not to be used as a learner, but to instead post process a NeuralNetLearner archive file and produce useful data for visualization of the state of the learner.

Parameters **filename** (*String*) – Filename of the GaussianProcessLearner archive.

Keyword Arguments **file_type** (*String*) – Can be 'mat' for matlab, 'pkl' for pickle or 'txt' for text. Default 'pkl'.

do_cross_sections(*upload*)

Produce a figure of the cross section about best cost and parameters

plot_density_surface()

Produce a density plot of the cost surface (only works when there are 2 parameters)

plot_losses()

Produce a figure of the loss as a function of training run.

plot_surface()

Produce a figure of the cost surface (only works when there are 2 parameters)

return_cross_sections(*points=100*, *cross_section_center=None*)

Finds the predicted global minima, then returns a list of vectors of parameters values, costs and uncertainties, corresponding to the 1D cross sections along each parameter axis through the predicted global minima.

Keyword Arguments

- **points** (*int*) – the number of points to sample along each cross section. Default value is 100.
- **cross_section_center** (*array*) – parameter array where the centre of the cross section should be taken. If None, the parameters for the best returned cost are used.

Returns a tuple (cross_arrays, cost_arrays, uncer_arrays) cross_parameter_arrays (list): a list of arrays for each cross section, with the values of the varied parameter going from the minimum to maximum value. cost_arrays (list): a list of arrays for the costs evaluated along each cross section about the minimum. uncertainty_arrays (list): a list of uncertainties

run()
Overides the GaussianProcessLearner multiprocessor run routine. Does nothing but makes a warning.

mloop.visualizations.**_color_from_controller_name**(*controller_name*)
Gives a color (as a number betweeen zero an one) corresponding to each controller name string.

mloop.visualizations.**_color_list_from_num_of_params**(*num_of_params*)
Gives a list of colors based on the number of parameters.

mloop.visualizations.**configure_plots**()
Configure the setting for the plots.

mloop.visualizations.**create_controller_visualizations**(*filename*, *file_type='pkl'*, *plot_cost_vs_run=True*, *plot_parameters_vs_run=True*, *plot_parameters_vs_cost=True*)
Runs the plots for a controller file.

Parameters **filename** (*Optional [string]*) – Filename for the controller archive.

Keyword Arguments

- **file_type** (*Optional [string]*) – File type 'pkl' pickle, 'mat' matlab or 'txt' text.
- **plot_cost_vs_run** (*Optional [bool]*) – If True plot cost versus run number, else do not. Default True.
- **plot_parameters_vs_run** (*Optional [bool]*) – If True plot parameters versus run number, else do not. Default True.
- **plot_parameters_vs_cost** (*Optional [bool]*) – If True plot parameters versus cost number, else do not. Default True.

mloop.visualizations.**create_differential_evolution_learner_visualizations**(*filename*, *file_type='pkl'*, *plot_params_vs_generat* *plot_costs_vs_generatio*
Runs the plots from a differential evolution learner file.

Parameters **filename** (*Optional [string]*) – Filename for the differential evolution archive. Must provide datetime or filename. Default None.

Keyword Arguments

- **file_type** (*Optional [string]*) – File type 'pkl' pickle, 'mat' matlab or 'txt' text.
- **plot_params_generations** (*Optional [bool]*) – If True plot parameters vs generations, else do not. Default True.
- **plot_costs_generations** (*Optional [bool]*) – If True plot costs vs generations, else do not. Default True.

mloop.visualizations.**create_gaussian_process_learner_visualizations**(*filename*, *file_type='pkl'*, *plot_cross_sections=True*, *plot_hyperparameters_vs_run=T*
Runs the plots from a gaussian process learner file.

Parameters **filename** (*Optional [string]*) – Filename for the gaussian process archive. Must provide datetime or filename. Default None.

Keyword Arguments

- **file_type** (*Optional [string]*) – File type 'pkl' pickle, 'mat' matlab or 'txt' text.

- **`plot_cross_sections`** (*`Optional [bool]`*) – If True plot predict landscape cross sections, else do not. Default True.

`mloop.visualizations.`**`create_neural_net_learner_visualizations`**(*filename*, *file_type='pkl'*, *plot_cross_sections=True*, *upload_cross_sections=False*)

Creates plots from a neural nets learner file.

Parameters **`filename`** (*`Optional [string]`*) – Filename for the neural net archive. Must provide datetime or filename. Default None.

Keyword Arguments

- **`file_type`** (*`Optional [string]`*) – File type 'pkl' pickle, 'mat' matlab or 'txt' text.
- **`plot_cross_sections`** (*`Optional [bool]`*) – If True plot predict landscape cross sections, else do not. Default True.

`mloop.visualizations.`**`show_all_default_visualizations`**(*controller*, *show_plots=True*)

Plots all visualizations available for a controller, and it's internal learners.

Parameters **`controller`** (`Controller`) – The controller to extract plots from

Keyword Arguments **`show_plots`** (`Controller`) – Determine whether to run plt.show() at the end or not. For debugging.

`mloop.visualizations.`**`show_all_default_visualizations_from_archive`**(*controller_filename*, *learner_filename*, *controller_type*, *show_plots=True*, *upload_cross_sections=False*)

CHAPTER 3

Indices

- genindex
- modindex
- search

Python Module Index

Index