PySwarms Documentation

Release 1.0.2

Lester James V. Miranda

May 01, 2019

General

It is intended for swarm intelligence researchers, practitioners, and students who prefer a high-level declarative interface for implementing PSO in their problems. PySwarms enables basic optimization with PSO and interaction with swarm optimizations. Check out more features below!

- **Free software:** MIT license
- **Github repository:** https://github.com/ljvmiranda921/pyswarms
- **Python versions:** 3.5 and 3.6

CHAPTER 1

Launching pad

- If you don't know what Particle Swarm Optimization is, read up this short Introduction! Then, if you plan to use PySwarms in your project, check the Installation guide and use-case examples.
- If you are a researcher in the field of swarm intelligence, and would like to include your technique in our list of optimizers, check our contributing page to see how to implement your optimizer using the current base classes in the library.
- If you are an open-source contributor, and would like to help PySwarms grow, be sure to check our Issues page in Github, and see the open issues with the tag help-wanted. Moreover, we accommodate contributions from first-time contributors! Just check our first-timers-only tag for open issues (Don't worry! We're happy to help you make your first PR!).

1.1 Introduction

1.1.1 It's all a treasure hunt

Imagine that you and your friends are looking for a treasure together. The treasure is magical, and it rewards not only the one who finds it, but also those near to it. Your group knows, approximately, where the treasure is, but not exactly sure of its definite location.

Your group then decided to split up with walkie-talkies and metal detectors. You use your walkie-talkie to inform everyone of your current position, and the metal detector to check your proximity to the treasure. In return, you gain knowledge of your friends' positions, and also their distance from the treasure.

As a member of the group, you have two options:

- Ignore your friends, and just search for the treasure the way you want it. Problem is, if you didn't find it, and you're far away from it, you get a very low reward.
- Using the information you gather from your group, coordinate and find the treasure together. The best way is to know who is the one nearest to the treasure, and move towards that person.

Here, it is evident that by using the information you can gather from your friends, you can increase the chances of finding the treasure, and at the same time maximize the group's reward. This is the basics of Particle Swarm Optimization (PSO). The group is called the *swarm*, you are a *particle*, and the treasure is the *global optimum [CI2007]*.

1.1.2 Particle Swarm Optimization (PSO)

As with the treasure example, the idea of PSO is to emulate the social behaviour of birds and fishes by initializing a set of candidate solutions to search for an optima. Particles are scattered around the search-space, and they move around it to find the position of the optima. Each particle represents a candidate solution, and their movements are affected in a two-fold manner: (1) their cognitive desire to search individually, (2) and the collective action of the group or its neighbors. It is a fairly simple concept with profound applications.

One interesting characteristic of PSO is that it does not use the gradient of the function, thus, objective functions need not to be differentiable. Moreover, the basic PSO is astonishingly simple. Adding variants to the original implementation can help it adapt to more complicated problems.

The original PSO algorithm is attributed to Eberhart, Kennedy, and Shi *[IJCNN1995] [ICEC2008]*. Nowadays, a lot of variations in topology, search-space characteristic, constraints, objectives, are being researched upon to solve a variety of problems.

1.1.3 Why make PySwarms?

In one of my graduate courses during Masters, my professor asked us to implement PSO for training a neural network. It was, in all honesty, my first experience of implementing an algorithm from concept to code. I found the concept of PSO very endearing, primarily because it gives us an insight on the advantage of collaboration given a social situation.

When I revisited my course project, I realized that PSO, given enough variations, can be used to solve a lot of problems: from simple optimization, to robotics, and to job-shop scheduling. I then decided to build a research toolkit that can be extended by the community (us!) and be used by anyone.

References

1.2 Features

1.2.1 Single-Objective Optimizers

These are standard optimization techniques for finding the optima of a single objective function.

Continuous

Single-objective optimization where the search-space is continuous. Perfect for optimizing various common functions.

- *pyswarms.single.global_best* - classic global-best Particle Swarm Optimization algorithm with a star-topology. Every particle compares itself with the best-performing particle in the swarm.
- *pyswarms.single.local_best* - classic local-best Particle Swarm Optimization algorithm with a ring-topology. Every particle compares itself only with its nearest-neighbours as computed by a distance metric.
- *pyswarms.single.general_optimizer* - alterable but still classic Particle Swarm Optimization algorithm with a custom topology. Every topology in the `pyswarms.backend` module can be passed as an argument.

Discrete

Single-objective optimization where the search-space is discrete. Useful for job-scheduling, traveling salesman, or any other sequence-based problems.

- *pyswarms.discrete.binary* - classic binary Particle Swarm Optimization algorithm without mutation. Uses a ring topology to choose its neighbours (but can be set to global).

1.2.2 Utilities

Benchmark Functions

These functions can be used as benchmarks for assessing the performance of the optimization algorithm.

- *pyswarms.utils.functions.single_obj* - single-objective test functions

Search

These search methods can be used to compare the relative performance of hyperparameter value combinations in reducing a specified objective function.

- *pyswarms.utils.search.grid_search* - exhaustive search of optimal performance on selected objective function over cartesian products of provided hyperparameter values
- *pyswarms.utils.search.random_search* - search for optimal performance on selected objective function over combinations of randomly selected hyperparameter values within specified bounds for specified number of selection iterations

Plotters

A quick and easy to use tool for the visualization of optimizations. It allows you to easily create animations and to visually check your optimization!

- *pyswarms.utils.plotters*

Environment

Deprecated since version 0.4.0: Use *pyswarms.utils.plotters* instead!

Various environments that allow you to analyze your swarm performance and make visualizations!

- `pyswarms.utils.environments.plot_environment` - an environment for plotting the cost history and animating particles in a 2D or 3D space.

1.3 Installation

1.3.1 Stable release

To install PySwarms, run this command in your terminal:

```
$ pip install pyswarms
```

This is the preferred method to install PySwarms, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

1.3.2 From sources

The sources for PySwarms can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/ljvmiranda921/pyswarms
```

Or download the tarball:

```
$ curl  -OL https://github.com/ljvmiranda921/pyswarms/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

1.4 Credits

This project was inspired by the pyswarm module that performs PSO with constrained support. The package was created with Cookiecutter and the audreyr/cookiecutter-pypackage project template.

1.4.1 Maintainers

- Lester James V. Miranda (@ljvmiranda921)
- Aaron Moser (@whzup)
- Siobhán K. Cronin (@SioKCronin)

1.4.2 Contributors

- Carl-K (@Carl-K)
- Andrew Jarcho (@jazcap53)
- Charalampos Papadimitriou (@CPapadim)
- Mamady Nabé (@mamadyonline)
- Erik (@slek120)
- Jay Speidell (@jayspeidell)
- Bradahoward (@bradahoward)
- Thomas (@ThomasCES)

1.5 History

1.5.1 0.1.0 (2017-07-12)

- First release on PyPI.
- **NEW:** Includes primary optimization techniques such as global-best PSO and local-best PSO - #1, #3

0.1.1 (2017-07-25)

- **FIX:** Patch on LocalBestPSO implementation. It seems that it's not returning the best value of the neighbors, this fixes the problem .
- **NEW:** Test functions for single-objective problems - #6, #10, #14. Contributed by @Carl-K. Thank you!

0.1.2 (2017-08-02)

- **NEW:** Binary Particle Swarm Optimization - #7, #17
- **FIX:** Fix on Ackley function return error - #22
- **IMPROVED:** Documentation and unit tests - #16

0.1.4 (2017-08-03)

- **FIX:** Added a patch to fix `pip` installation

0.1.5 (2017-08-11)

- **NEW:** easy graphics environment. This new plotting environment makes it easier to plot the costs and swarm movement in 2-d or 3-d planes - #30, #31

0.1.6 (2017-09-24)

- **NEW:** Native GridSearch and RandomSearch implementations for finding the best hyperparameters in controlling swarm behaviour - #4, #20, #25. Contributed by @SioKCronin. Thanks a lot!
- **NEW:** Added tests for hyperparameter search techniques - #27, #28, #40. Contributed by @jazcap53. Thank you so much!
- **IMPROVED:** Updated structure of Base classes for higher extensibility

0.1.7 (2017-09-25)

- **FIX:** Fixed patch on `local_best.py` and `binary.py` - #33, #34. Thanks for the awesome fix, @CPapadim!
- **NEW:** Git now ignores IPython notebook checkpoints

0.1.8 (2018-01-11)

- **NEW:** PySwarms is now published on the Journal of Open Source Software (JOSS)! You can check the review here. In addition, you can also find our paper in this link. Thanks a lot to @kyleniemeyer and @stsievert for the thoughtful reviews and comments.

0.1.9 (2018-04-20)

- **NEW:** You can now set the initial position wherever you want - #93
- **FIX:** Quick-fix for the Rosenbrock function - #98
- **NEW:** Tolerance can now be set to break during iteration - #100

Thanks for all the wonderful Pull Requests, @mamadyonline!

1.5.2 0.2.0 (2018-06-11)

- **NEW:** New PySwarms backend. You can now build native swarm implementations using this module! - #115, #116, #117
- **DEPRECATED:** Drop Python 2.7 version support. This package now supports Python 3.4 and up - #113
- **IMPROVED:** All tests were ported into pytest - #114

0.2.1 (2018-06-27)

- **FIX:** Fix sigmoid function in BinaryPSO - #145. Thanks a lot @ThomasCES!

1.5.3 0.3.0 (2018-08-10)

- **NEW:** New topologies: Pyramid, Random, and Von Neumann. More ways for your particles to interact! - #176, #177, #155, #142. Thanks a lot @whzup!
- **NEW:** New GeneralOptimizer algorithm that allows you to switch-out topologies for your optimization needs - #151. Thanks a lot @whzup!
- **NEW:** All topologies now have a static attribute. Neigbors can now be set initially or computed dynamically - #164. Thanks a lot @whzup!
- **NEW:** New single-objective functions - #168. Awesome work, @jayspeidell!
- **NEW:** New tutorial on Inverse Kinematics using Particle Swarm Optimization - #141. Thanks a lot @whzup!
- **NEW:** New plotters module for visualization. The environment module is now deprecated - #135
- **IMPROVED:** Keyword arguments can now be passed in the `optimize()` method for your custom objective functions - #144. Great job, @bradahoward

0.3.1 (2018-08-13)

- **NEW:** New collaboration tool using Vagrantfiles - #193. Thanks a lot @jdbohrman!
- **NEW:** Add configuration file for pyup.io - #210
- **FIX:** Fix incomplete documentation in ReadTheDocs - #208
- **IMPROVED:** Update dependencies via pyup - #204

1.5.4 0.4.0 (2019-01-29)

- **NEW:** The console output is now generated by the `Reporter` module - #227
- **NEW:** A `@cost` decorator which automatically scales to the whole swarm - #226
- **FIX:** A bug in the topologies where the best position in some topologies was not calculated using the nearest neighbours - #253
- **FIX:** Swarm init positions - #249 Thanks @dfhljf!
- **IMPROVED:** Better naming for the benchmark functions - #222 Thanks @nik1082!
- **IMPROVED:** Error handling in the `Optimizers` - #232
- **IMPROVED:** New management method for dependencies - #263
- **DEPRECATED:** The *environments* module is now deprecated - #217

1.5.5 1.0.0 (2019-02-08)

This is the first major release of PySwarms. Starting today, we will be adhering to a better semantic versioning guidelines. We will be updating the project wikis shortly after. The maintainers believe that PySwarms is mature enough to merit a version 1, this would also help us release more often (mostly minor releases) and create patch releases as soon as possible.

Also, we will be maintaining a quarterly release cycle, where the **next minor release (v.1.0.0) will be on June**. All enhancements and new features will be staged on the `development` branch, then will be merged back to the `master` branch at the end of the cycle. However, bug fixes and documentation errors will merit a patch release, and will be merged to `master` immediately.

- **NEW:** Boundary and velocity handlers to resolve stuck particles - #238 . All thanks for our maintainer, @whzup !
- **FIX:** Duplication function calls during optimization, hopefully your long-running objective functions won't take doubly long. - #266. Thank you @danielcorreia96 !

1.0.1 (2019-02-14)

- **FIX:** Handlers memory management so that it works all the time - #286 . Thanks for this @whzup !
- **FIX:** Re-introduce fix for multiple optimization function calls - #290 . Thank you once more @danielcorreia96 !

1.0.2 (2019-02-17)

- **FIX:** BinaryPSO should return final best position instead of final swarm - #293 . Thank you once more @danielcorreia96 !

1.6 Tutorials

Below are some examples describing how the PySwarms API works. If you wish to check the actual Jupyter Notebooks, please go to this link

1.6.1 Basic Optimization

In this example, we'll be performing a simple optimization of single-objective functions using the global-best optimizer in `pyswarms.single.GBestPSO` and the local-best optimizer in `pyswarms.single.LBestPSO`. This aims to demonstrate the basic capabilities of the library when applied to benchmark problems.

```
import sys
# Change directory to access the pyswarms module
sys.path.append('../')
```

```
print('Running on Python version: {}'.format(sys.version))
```

```
Running on Python version: 3.6.7 (default, Oct 22 2018, 11:32:17)
[GCC 8.2.0]
```

```
# Import modules
import numpy as np

# Import PySwarms
import pyswarms as ps
from pyswarms.utils.functions import single_obj as fx
```

(continues on next page)

(continued from previous page)

```
# Some more magic so that the notebook will reload external python modules;
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
```

Optimizing a function

First, let's start by optimizing the sphere function. Recall that the minima of this function can be located at `f(0,0..,0)` with a value of `0`. In case you don't remember the characteristics of a given function, simply call `help(<function>)`.

For now let's just set some arbitrary parameters in our optimizers. There are, at minimum, three steps to perform optimization:

1. Set the hyperparameters to configure the swarm as a `dict`.
2. Create an instance of the optimizer by passing the dictionary along with the necessary arguments.
3. Call the `optimize()` method and have it store the optimal cost and position in a variable.

The `optimize()` method returns a `tuple` of values, one of which includes the optimal cost and position after optimization. You can store it in a single variable and just index the values, or unpack it using several variables at once.

```
%%time
# Set-up hyperparameters
options = {'c1': 0.5, 'c2': 0.3, 'w':0.9}

# Call instance of PSO
optimizer = ps.single.GlobalBestPSO(n_particles=10, dimensions=2, options=options)

# Perform optimization
cost, pos = optimizer.optimize(fx.sphere, iters=1000)
```

```
2019-01-30 04:23:31,846 - pyswarms.single.global_best - INFO - Optimize for 1000␣
↪iters with {'c1': 0.5, 'c2': 0.3, 'w': 0.9}
pyswarms.global_best: 100%||1000/1000, best_cost=5.34e-43
2019-01-30 04:23:46,631 - pyswarms.single.global_best - INFO - Optimization␣
↪finished | best cost: 5.3409429804817095e-43, best pos: [-4.84855366e-22 -5.
↪46817677e-22]
```

```
CPU times: user 5.63 s, sys: 916 ms, total: 6.55 s
Wall time: 14.8 s
```

We can see that the optimizer was able to find a good minima as shown above. You can control the verbosity of the output using the `verbose` argument, and the number of steps to be printed out using the `print_step` argument.

Now, let's try this one using local-best PSO:

```
%%time
# Set-up hyperparameters
options = {'c1': 0.5, 'c2': 0.3, 'w':0.9, 'k': 2, 'p': 2}

# Call instance of PSO
optimizer = ps.single.LocalBestPSO(n_particles=10, dimensions=2, options=options)

# Perform optimization
cost, pos = optimizer.optimize(fx.sphere, iters=1000)
```

```
2019-01-30 04:23:46,672 - pyswarms.single.local_best - INFO - Optimize for 1000␣
↪iters with {'c1': 0.5, 'c2': 0.3, 'w': 0.9, 'k': 2, 'p': 2}
pyswarms.single.local_best: 100%||1000/1000, best_cost=1.19e-48
2019-01-30 04:24:02,254 - pyswarms.single.local_best - INFO - Optimization␣
↪finished | best cost: 1.1858559943008184e-48, best pos: [5.47013119e-24 7.
↪95177208e-25]
```

```
CPU times: user 6.63 s, sys: 1.04 s, total: 7.68 s
Wall time: 15.6 s
```

Optimizing a function with bounds

Another thing that we can do is to set some bounds into our solution, so as to contain our candidate solutions within a specific range. We can do this simply by passing a `bounds` parameter, of type `tuple`, when creating an instance of our swarm. Let's try this using the global-best PSO with the Rastrigin function (`rastrigin` in `pyswarms.utils.functions.single_obj`).

Recall that the Rastrigin function is bounded within `[-5.12, 5.12]`. If we pass an unbounded swarm into this function, then a `ValueError` might be raised. So what we'll do is to create a bound within the specified range. There are some things to remember when specifying a bound:

- A bound should be of type tuple with length 2.
- It should contain two `numpy.ndarrays` so that we have a `(min_bound, max_bound)`
- Obviously, all values in the `max_bound` should always be greater than the `min_bound`. Their shapes should match the dimensions of the swarm.

What we'll do now is to create a 10-particle, 2-dimensional swarm. This means that we have to set our maximum and minimum boundaries with the shape of 2. In case we want to initialize an n-dimensional swarm, we then have to set our bounds with the same shape n. A fast workaround for this would be to use the `numpy.ones` function multiplied by a constant.

```
# Create bounds
max_bound = 5.12 * np.ones(2)
min_bound = - max_bound
bounds = (min_bound, max_bound)
```

```
%%time
# Initialize swarm
options = {'c1': 0.5, 'c2': 0.3, 'w':0.9}

# Call instance of PSO with bounds argument
optimizer = ps.single.GlobalBestPSO(n_particles=10, dimensions=2, options=options,␣
↪bounds=bounds)

# Perform optimization
cost, pos = optimizer.optimize(fx.rastrigin, iters=1000)
```

```
2019-01-30 04:24:02,463 - pyswarms.single.global_best - INFO - Optimize for 1000␣
↪iters with {'c1': 0.5, 'c2': 0.3, 'w': 0.9}
pyswarms.single.global_best: 100%||1000/1000, best_cost=0
2019-01-30 04:24:17,995 - pyswarms.single.global_best - INFO - Optimization␣
↪finished | best cost: 0.0, best pos: [1.99965504e-09 9.50602717e-10]
```

```
CPU times: user 6.74 s, sys: 1.01 s, total: 7.75 s
Wall time: 15.5 s
```

Basic Optimization with Arguments

Here, we will run a basic optimization using an objective function that needs parameterization. We will use the `single.GBestPSO` and a version of the rosenbrock function to demonstrate

```
import sys
# change directory to access pyswarms
sys.path.append('../')

print("Running Python {}".format(sys.version))
```

```
Running Python 3.6.7 (default, Oct 22 2018, 11:32:17)
[GCC 8.2.0]
```

```
# import modules
import numpy as np

# create a parameterized version of the classic Rosenbrock unconstrained␣
↪optimzation function
def rosenbrock_with_args(x, a, b, c=0):

    f = (a - x[:, 0]) ** 2 + b * (x[:, 1] - x[:, 0] ** 2) ** 2 + c
    return f
```

Using Arguments

Arguments can either be passed in using a tuple or a dictionary, using the `kwargs={}` paradigm. First lets optimize the Rosenbrock function using keyword arguments. Note in the definition of the Rosenbrock function above, there were two arguments that need to be passed other than the design variables, and one optional keyword argument, `a`, `b`, and `c`, respectively

```
from pyswarms.single.global_best import GlobalBestPSO

# instantiate the optimizer
x_max = 10 * np.ones(2)
x_min = -1 * x_max
bounds = (x_min, x_max)
options = {'c1': 0.5, 'c2': 0.3, 'w': 0.9}
optimizer = GlobalBestPSO(n_particles=10, dimensions=2, options=options,␣
↪bounds=bounds)

# now run the optimization, pass a=1 and b=100 as a tuple assigned to args

cost, pos = optimizer.optimize(rosenbrock_with_args, 1000, a=1, b=100, c=0)
```

```
2019-01-30 04:24:18,385 - pyswarms.single.global_best - INFO - Optimize for 1000␣
↪iters with {'c1': 0.5, 'c2': 0.3, 'w': 0.9}
pyswarms.single.global_best: 100%||1000/1000, best_cost=1.65e-18
2019-01-30 04:24:33,873 - pyswarms.single.global_best - INFO - Optimization␣
↪finished | best cost: 1.6536536065757395e-18, best pos: [1. 1.]
```

It is also possible to pass a dictionary of key word arguments by using ** decorator when passing the dict

```
kwargs={"a": 1.0, "b": 100.0, 'c':0}
cost, pos = optimizer.optimize(rosenbrock_with_args, 1000, **kwargs)
```

```
2019-01-30 04:24:33,904 - pyswarms.single.global_best - INFO - Optimize for 1000␣
↪iters with {'c1': 0.5, 'c2': 0.3, 'w': 0.9}
pyswarms.single.global_best: 100%||1000/1000, best_cost=9.13e-19
```

(continues on next page)

(continued from previous page)

```
2019-01-30 04:24:49,482 - pyswarms.single.global_best - INFO - Optimization␣
↪finished | best cost: 9.132114249459913e-19, best pos: [1. 1.]
```

Any key word arguments in the objective function can be left out as they will be passed the default as defined in the prototype. Note here, `c` is not passed into the function.

```
cost, pos = optimizer.optimize(rosenbrock_with_args, 1000, a=1, b=100)
```

```
2019-01-30 04:24:49,518 - pyswarms.single.global_best - INFO - Optimize for 1000␣
↪iters with {'c1': 0.5, 'c2': 0.3, 'w': 0.9}
pyswarms.single.global_best: 100%||1000/1000, best_cost=9.13e-19
2019-01-30 04:25:05,071 - pyswarms.single.global_best - INFO - Optimization␣
↪finished | best cost: 9.125748012380431e-19, best pos: [1. 1.]
```

1.6.2 Training a Neural Network

In this example, we'll be training a neural network using particle swarm optimization. For this we'll be using the standard global-best PSO `pyswarms.single.GBestPSO` for optimizing the network's weights and biases. This aims to demonstrate how the API is capable of handling custom-defined functions.

For this example, we'll try to classify the three iris species in the Iris Dataset.

```
# Import modules
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris


# Import PySwarms
import pyswarms as ps

# Some more magic so that the notebook will reload external python modules;
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
```

First, we'll load the dataset from `scikit-learn`. The Iris Dataset contains 3 classes for each of the iris species (*iris setosa*, *iris virginica*, and *iris versicolor*). It has 50 samples per class with 150 samples in total, making it a very balanced dataset. Each sample is characterized by four features (or dimensions): sepal length, sepal width, petal length, petal width.

```
# Load the iris dataset
data = load_iris()

# Store the features as X and the labels as y
X = data.data
y = data.target
```

Constructing a custom objective function

Recall that neural networks can simply be seen as a mapping function from one space to another. For now, we'll build a simple neural network with the following characteristics:

- Input layer size: 4
- Hidden layer size: 20 (activation: $\tanh(x)$)
- Output layer size: 3 (activation: $softmax(x)$)

Things we'll do:

1. Create a `forward_prop` method that will do forward propagation for one particle.
2. Create an overhead objective function `f()` that will compute `forward_prop()` for the whole swarm.

What we'll be doing then is to create a swarm with a number of dimensions equal to the weights and biases. We will **unroll** these parameters into an n-dimensional array, and have each particle take on different values. Thus, each particle represents a candidate neural network with its own weights and bias. When feeding back to the network, we will reconstruct the learned weights and biases.

When rolling-back the parameters into weights and biases, it is useful to recall the shape and bias matrices:

- Shape of input-to-hidden weight matrix: (4, 20)
- Shape of input-to-hidden bias array: (20,)
- Shape of hidden-to-output weight matrix: (20, 3)
- Shape of hidden-to-output bias array: (3,)

By unrolling them together, we have $(4 * 20) + (20 * 3) + 20 + 3 = 163$ parameters, or 163 dimensions for each particle in the swarm.

The negative log-likelihood will be used to compute for the error between the ground-truth values and the predictions. Also, because PSO doesn't rely on the gradients, we'll not be performing backpropagation (this may be a good thing or bad thing under some circumstances).

Now, let's write the forward propagation procedure as our objective function. Let X be the input, z_l the pre-activation at layer l, and a_l the activation for layer l:

```
# Forward propagation
def forward_prop(params):
    """Forward propagation as objective function

    This computes for the forward propagation of the neural network, as
    well as the loss. It receives a set of parameters that must be
    rolled-back into the corresponding weights and biases.

    Inputs
    ------
    params: np.ndarray
        The dimensions should include an unrolled version of the
        weights and biases.

    Returns
    -------
    float
        The computed negative log-likelihood loss given the parameters
    """
    # Neural network architecture
    n_inputs = 4
    n_hidden = 20
    n_classes = 3

    # Roll-back the weights and biases
    W1 = params[0:80].reshape((n_inputs,n_hidden))
    b1 = params[80:100].reshape((n_hidden,))
    W2 = params[100:160].reshape((n_hidden,n_classes))
    b2 = params[160:163].reshape((n_classes,))

    # Perform forward propagation
    z1 = X.dot(W1) + b1  # Pre-activation in Layer 1
    a1 = np.tanh(z1)     # Activation in Layer 1
    z2 = a1.dot(W2) + b2 # Pre-activation in Layer 2
    logits = z2          # Logits for Layer 2
```

(continues on next page)

(continued from previous page)

```
    # Compute for the softmax of the logits
    exp_scores = np.exp(logits)
    probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)

    # Compute for the negative log likelihood
    N = 150 # Number of samples
    corect_logprobs = -np.log(probs[range(N), y])
    loss = np.sum(corect_logprobs) / N

    return loss
```

Now that we have a method to do forward propagation for one particle (or for one set of dimensions), we can then create a higher-level method to compute `forward_prop()` to the whole swarm:

```
def f(x):
    """Higher-level method to do forward_prop in the
    whole swarm.

    Inputs
    ------
    x: numpy.ndarray of shape (n_particles, dimensions)
        The swarm that will perform the search

    Returns
    -------
    numpy.ndarray of shape (n_particles, )
        The computed loss for each particle
    """
    n_particles = x.shape[0]
    j = [forward_prop(x[i]) for i in range(n_particles)]
    return np.array(j)
```

Performing PSO on the custom-function

Now that everything has been set-up, we just call our global-best PSO and run the optimizer as usual. For now, we'll just set the PSO parameters arbitrarily.

```
# Initialize swarm
options = {'c1': 0.5, 'c2': 0.3, 'w':0.9}

# Call instance of PSO
dimensions = (4 * 20) + (20 * 3) + 20 + 3
optimizer = ps.single.GlobalBestPSO(n_particles=100, dimensions=dimensions,␣
↪options=options)

# Perform optimization
cost, pos = optimizer.optimize(f, print_step=100, iters=1000, verbose=3)
```

```
Iteration 1/1000, cost: 1.09858937026
Iteration 101/1000, cost: 0.0516382653768
Iteration 201/1000, cost: 0.0416398234107
Iteration 301/1000, cost: 0.0399519086999
Iteration 401/1000, cost: 0.0396579575634
Iteration 501/1000, cost: 0.0394155032472
Iteration 601/1000, cost: 0.0388702854787
Iteration 701/1000, cost: 0.0386106261126
Iteration 801/1000, cost: 0.0384067695633
Iteration 901/1000, cost: 0.0370548470526
```

(continues on next page)

(continued from previous page)

```
==============================
Optimization finished!
Final cost: 0.0362
Best value: 0.170569 -4.586860 -0.726267 -3.602894 0.085438 -3.167099 ...
```

Checking the accuracy

We can then check the accuracy by performing forward propagation once again to create a set of predictions. Then it's only a simple matter of matching which one's correct or not. For the `logits`, we take the `argmax`. Recall that the softmax function returns probabilities where the whole vector sums to 1. We just take the one with the highest probability then treat it as the network's prediction.

Moreover, we let the best position vector found by the swarm be the weight and bias parameters of the network.

```
def predict(X, pos):
    """
    Use the trained weights to perform class predictions.

    Inputs
    ------
    X: numpy.ndarray
        Input Iris dataset
    pos: numpy.ndarray
        Position matrix found by the swarm. Will be rolled
        into weights and biases.
    """
    # Neural network architecture
    n_inputs = 4
    n_hidden = 20
    n_classes = 3

    # Roll-back the weights and biases
    W1 = pos[0:80].reshape((n_inputs,n_hidden))
    b1 = pos[80:100].reshape((n_hidden,))
    W2 = pos[100:160].reshape((n_hidden,n_classes))
    b2 = pos[160:163].reshape((n_classes,))

    # Perform forward propagation
    z1 = X.dot(W1) + b1  # Pre-activation in Layer 1
    a1 = np.tanh(z1)     # Activation in Layer 1
    z2 = a1.dot(W2) + b2 # Pre-activation in Layer 2
    logits = z2          # Logits for Layer 2

    y_pred = np.argmax(logits, axis=1)
    return y_pred
```

And from this we can just compute for the accuracy. We perform predictions, compare an equivalence to the ground-truth value `y`, and get the mean.

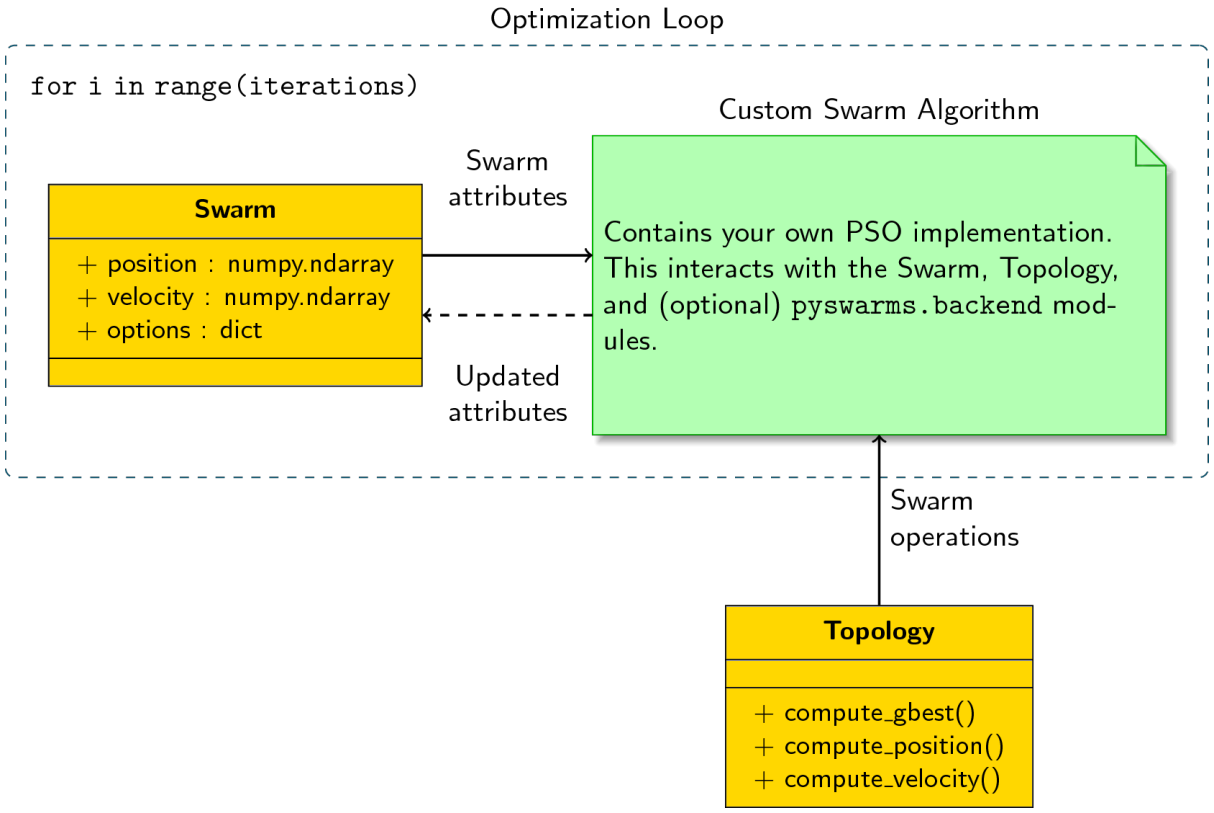
```
(predict(X, pos) == y).mean()
```

```
0.9866666666666669
```

1.6.3 Writing your own optimization loop

In this example, we will use the `pyswarms.backend` module to write our own optimization loop. We will try to recreate the Global best PSO using the native backend in PySwarms. Hopefully, this short tutorial can give you

an idea on how to use this for your own custom swarm implementation. The idea is simple, again, let's refer to this diagram:



Some things to note:

- Initialize a `Swarm` class and update its attributes for every iteration.
- Initialize a `Topology` class (in this case, we'll use a `Star` topology), and use its methods to operate on the Swarm.
- We can also use some additional methods in `pyswarms.backend` depending on our needs.

Thus, for each iteration: 1. We take an attribute from the `Swarm` class. 2. Operate on it according to our custom algorithm with the help of the `Topology` class; and 3. Update the `Swarm` class with the new attributes.

```
# Change directory to access the pyswarms module
sys.path.append('../')
```

```
print('Running on Python version: {}'.format(sys.version))
```

```
Running on Python version: 3.6.7 (default, Oct 22 2018, 11:32:17)
[GCC 8.2.0]
```

```
# Import modules
import numpy as np

# Import sphere function as objective function
from pyswarms.utils.functions.single_obj import sphere as f

# Import backend modules
import pyswarms.backend as P
from pyswarms.backend.topology import Star

# Some more magic so that the notebook will reload external python modules;
```

(continues on next page)

(continued from previous page)

```
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
```

Native global-best PSO implementation

Now, the global best PSO pseudocode looks like the following (adapted from A. Engelbrecht, "Computational Intelligence: An Introduction, 2002):

```
# Python-version of gbest algorithm from Engelbrecht's book
for i in range(iterations):
    for particle in swarm:
        # Part 1: If current position is less than the personal best,
        if f(current_position[particle]) < f(personal_best[particle]):
            # Update personal best
            personal_best[particle] = current_position[particle]
        # Part 2: If personal best is less than global best,
        if f(personal_best[particle]) < f(global_best):
            # Update global best
            global_best = personal_best[particle]
        # Part 3: Update velocity and position matrices
        update_velocity()
        update_position()
```

As you can see, the standard PSO has a three-part scheme: update the personal best, update the global best, and update the velocity and position matrices. We'll follow this three part scheme in our native implementation using the PySwarms backend

Let's make a 2-dimensional swarm with 50 particles that will optimize the sphere function. First, let's initialize the important attributes in our algorithm

```
my_topology = Star() # The Topology Class
my_options = {'c1': 0.6, 'c2': 0.3, 'w': 0.4} # arbitrarily set
my_swarm = P.create_swarm(n_particles=50, dimensions=2, options=my_options) # The␣
↪Swarm Class

print('The following are the attributes of our swarm: {}'.format(my_swarm.__dict__.
↪keys()))
```

```
The following are the attributes of our swarm: dict_keys(['position', 'velocity',
↪'n_particles', 'dimensions', 'options', 'pbest_pos', 'best_pos', 'pbest_cost',
↪'best_cost', 'current_cost'])
```

Now, let's write our optimization loop!

```
iterations = 100 # Set 100 iterations
for i in range(iterations):
    # Part 1: Update personal best
    my_swarm.current_cost = f(my_swarm.position) # Compute current cost
    my_swarm.pbest_cost = f(my_swarm.pbest_pos)  # Compute personal best pos
    my_swarm.pbest_pos, my_swarm.pbest_cost = P.compute_pbest(my_swarm) # Update␣
↪and store

    # Part 2: Update global best
    # Note that gbest computation is dependent on your topology
    if np.min(my_swarm.pbest_cost) < my_swarm.best_cost:
        my_swarm.best_pos, my_swarm.best_cost = my_topology.compute_gbest(my_swarm)

    # Let's print our output
```

(continues on next page)

(continued from previous page)

```
    if i%20==0:
        print('Iteration: {} | my_swarm.best_cost: {:.4f}'.format(i+1, my_swarm.
↪best_cost))

    # Part 3: Update position and velocity matrices
    # Note that position and velocity updates are dependent on your topology
    my_swarm.velocity = my_topology.compute_velocity(my_swarm)
    my_swarm.position = my_topology.compute_position(my_swarm)

print('The best cost found by our swarm is: {:.4f}'.format(my_swarm.best_cost))
print('The best position found by our swarm is: {}'.format(my_swarm.best_pos))
```

```
Iteration: 1 | my_swarm.best_cost: 0.0020
Iteration: 21 | my_swarm.best_cost: 0.0000
Iteration: 41 | my_swarm.best_cost: 0.0000
Iteration: 61 | my_swarm.best_cost: 0.0000
Iteration: 81 | my_swarm.best_cost: 0.0000
The best cost found by our swarm is: 0.0000
The best position found by our swarm is: [ 1.26773865e-17 -1.24781239e-18]
```

Of course, we can just use the `GlobalBestPSO` implementation in PySwarms (it has boundary support, tolerance, initial positions, etc.):

```
from pyswarms.single import GlobalBestPSO

optimizer = GlobalBestPSO(n_particles=50, dimensions=2, options=my_options) #␣
↪Reuse our previous options
optimizer.optimize(f, iters=100)
```

```
2019-01-30 23:50:06,728 - pyswarms.single.global_best - INFO - Optimize for 100␣
↪iters with {'c1': 0.6, 'c2': 0.3, 'w': 0.4}
pyswarms.single.global_best: 100%||100/100, best_cost=0.00293
2019-01-30 23:50:08,269 - pyswarms.single.global_best - INFO - Optimization␣
↪finished | best cost: 0.0029270203924585485, best pos: [0.0497835  0.02118073]
```

```
(0.0029270203924585485, array([0.0497835 , 0.02118073]))
```

1.6.4 Visualization

PySwarms implements tools for visualizing the behavior of your swarm. These are built on top of `matplotlib`, thus rendering charts that are easy to use and highly-customizable. However, it must be noted that in order to use the animation capability in PySwarms (and in `matplotlib` for that matter), at least one writer tool must be installed. Some available tools include: * ffmpeg * ImageMagick * MovieWriter (base)

In the following demonstration, the `ffmpeg` tool is used. For Linux and Windows users, it can be installed via:

```
$ conda install -c conda-forge ffmpeg
```

In this example, we will demonstrate three plotting methods available on PySwarms: - `plot_cost_history`: for plotting the cost history of a swarm given a matrix - `plot_contour`: for plotting swarm trajectories of a 2D-swarm in two-dimensional space - `plot_surface`: for plotting swarm trajectories of a 2D-swarm in three-dimensional space

```
# Import modules
import matplotlib.pyplot as plt
import numpy as np
from matplotlib import animation, rc
from IPython.display import HTML
```

(continues on next page)

(continued from previous page)

```
# Import PySwarms
import pyswarms as ps
from pyswarms.utils.functions import single_obj as fx
from pyswarms.utils.plotters import (plot_cost_history, plot_contour, plot_surface)

# Some more magic so that the notebook will reload external python modules;
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
```

The first step is to create an optimizer. Here, we're going to use Global-best PSO to find the minima of a sphere function. As usual, we simply create an instance of its class `pyswarms.single.GlobalBestPSO` by passing the required parameters that we will use. Then, we'll call the `optimize()` method for 100 iterations.

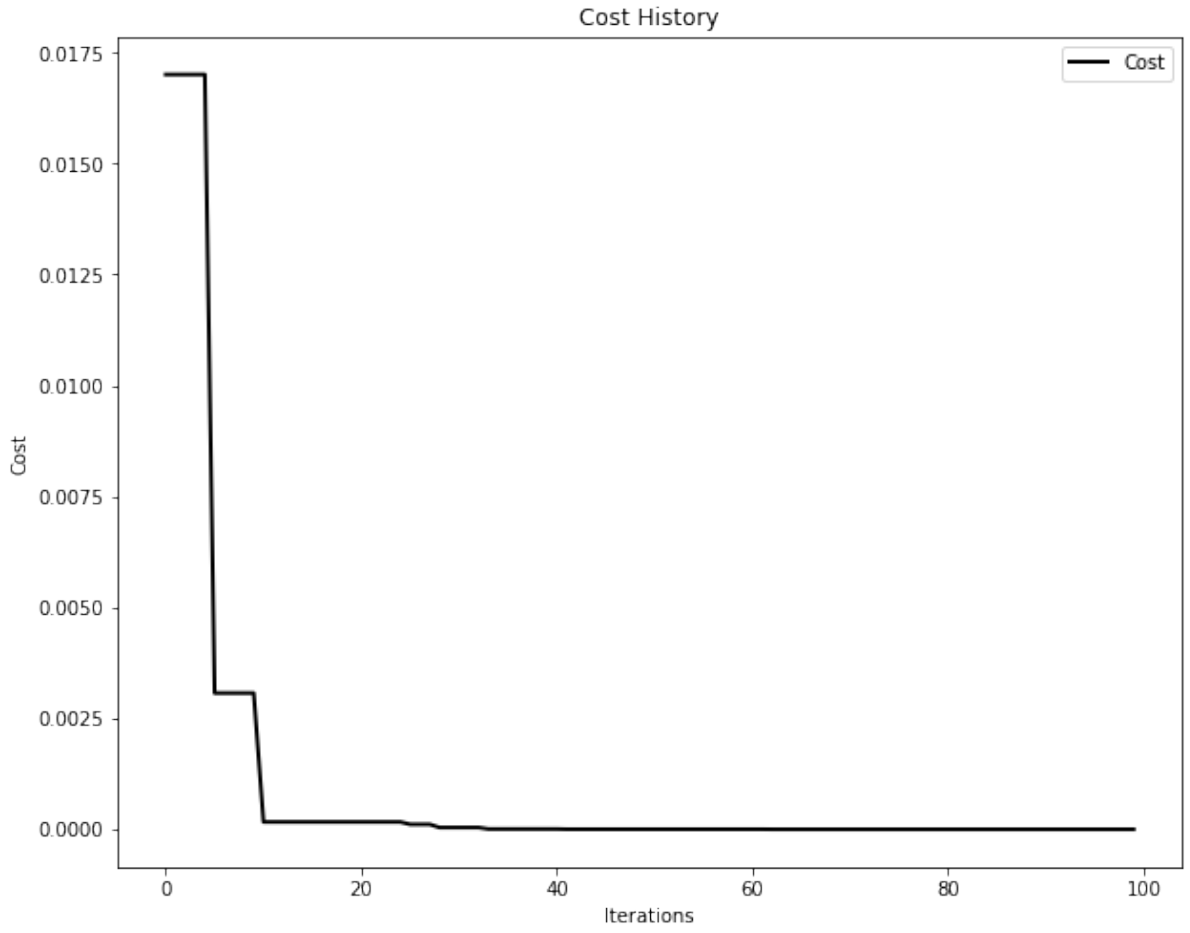
```
options = {'c1':0.5, 'c2':0.3, 'w':0.9}
optimizer = ps.single.GlobalBestPSO(n_particles=50, dimensions=2, options=options)
cost, pos = optimizer.optimize(fx.sphere, iters=100)
```

Plotting the cost history

To plot the cost history, we simply obtain the `cost_history` from the `optimizer` class and pass it to the `cost_history` function. Furthermore, this method also accepts a keyword argument `**kwargs` similar to `matplotlib`. This enables us to further customize various artists and elements in the plot. In addition, we can obtain the following histories from the same class:

- mean_neighbor_history: average local best history of all neighbors throughout optimization
- mean_pbest_history: average personal best of the particles throughout optimization

```
plot_cost_history(cost_history=optimizer.cost_history)
plt.show()
```



Animating swarms

The `plotters` module offers two methods to perform animation, `plot_contour()` and `plot_surface()`. As its name suggests, these methods plot the particles in a 2-D or 3-D space.

Each animation method returns a `matplotlib.animation.Animation` class that still needs to be animated by a `Writer` class (thus necessitating the installation of a writer module). For the proceeding examples, we will convert the animations into an HTML5 video. In such case, we need to invoke some extra methods to do just that.

```
# equivalent to rcParams['animation.html'] = 'html5'
# See http://louistiao.me/posts/notebooks/save-matplotlib-animations-as-gifs/
rc('animation', html='html5')
```

Lastly, it would be nice to add meshes in our swarm to plot the sphere function. This enables us to visually recognize where the particles are with respect to our objective function. We can accomplish that using the `Mesher` class.

```
from pyswarms.utils.plotters.formatters import Mesher

# Initialize mesher with sphere function
m = Mesher(func=fx.sphere)
```

There are different formatters available in the `pyswarms.utils.plotters.formatters` module to customize your plots and visualizations. Aside from `Mesher`, there is a `Designer` class for customizing font sizes, figure sizes, etc. and an `Animator` class to set delays and repeats during animation.

Plotting in 2-D space

We can obtain the swarm's position history using the `pos_history` attribute from the `optimizer` instance. To plot a 2D-contour, simply pass this together with the `Mesher` to the `plot_contour()` function. In addition, we can also mark the global minima of the sphere function, `(0,0)`, to visualize the swarm's "target".

```
# Make animation
animation = plot_contour(pos_history=optimizer.pos_history,
                         mesher=m,
                         mark=(0,0))

# Enables us to view it in a Jupyter notebook
HTML(animation.to_html5_video())
```

Plotting in 3-D space

To plot in 3D space, we need a position-fitness matrix with shape `(iterations, n_particles, 3)`. The first two columns indicate the x-y position of the particles, while the third column is the fitness of that given position. You need to set this up on your own, but we have provided a helper function to compute this automatically

```
# Obtain a position-fitness matrix using the Mesher.compute_history_3d()
# method. It requires a cost history obtainable from the optimizer class
pos_history_3d = m.compute_history_3d(optimizer.pos_history)

# Make a designer and set the x,y,z limits to (-1,1), (-1,1) and (-0.1,1)␣
↪respectively
from pyswarms.utils.plotters.formatters import Designer
d = Designer(limits=[(-1,1), (-1,1), (-0.1,1)], label=['x-axis', 'y-axis', 'z-axis
↪'])

# Make animation
animation3d = plot_surface(pos_history=pos_history_3d, # Use the cost_history we␣
↪computed
                           mesher=m, designer=d,       # Customizations
                           mark=(0,0,0))               # Mark minima

# Enables us to view it in a Jupyter notebook
HTML(animation3d.to_html5_video())
```

1.7 Use-cases

Below are some examples on how to use PSO in different applications. If you wish to check the actual Jupyter Notebooks, please go to this link

1.7.1 Feature Subset Selection

In this example, we'll be using the optimizer `pyswarms.discrete.BinaryPSO` to perform feature subset selection to improve classifier performance. But before we jump right on to the coding, let's first explain some relevant concepts:

A short primer on feature selection

The idea for feature subset selection is to be able to find the best features that are suitable to the classification task. We must understand that not all features are created equal, and some may be more relevant than others. Thus, if we're given an array of features, how can we know the most optimal subset? (yup, this is a rhetorical question!)

For a Binary PSO, the position of the particles are expressed in two terms: `1` or `0` (or on and off). If we have a particle x on d-dimensions, then its position can be defined as:

$$x = [x_1, x_2, x_3, \ldots, x_d] \quad \text{where} \quad x_i \in 0, 1$$

In this case, the position of the particle for each dimension can be seen as a simple matter of on and off.

Feature selection and the objective function

Now, suppose that we're given a dataset with d features. What we'll do is that we're going to *assign each feature as a dimension of a particle*. Hence, once we've implemented Binary PSO and obtained the best position, we can then interpret the binary array (as seen in the equation above) simply as turning a feature on and off.

As an example, suppose we have a dataset with 5 features, and the final best position of the PSO is:

```
>>> optimizer.best_pos
np.array([0, 1, 1, 1, 0])
>>> optimizer.best_cost
0.00
```

Then this means that the second, third, and fourth (or first, second, and third in zero-index) that are turned on are the selected features for the dataset. We can then train our classifier using only these features while dropping the others. How do we then define our objective function? (Yes, another rhetorical question!). We can design our own, but for now I'll be taking an equation from the works of Vieira, Mendoca, Sousa, et al. (2013).

$$f(X) = \alpha(1 - P) + (1 - \alpha)\left(1 - \frac{N_f}{N_t}\right)$$

Where α is a hyperparameter that decides the tradeoff between the classifier performance P, and the size of the feature subset N_f with respect to the total number of features N_t. The classifier performance can be the accuracy, F-score, precision, and so on.

```
# Import modules
import numpy as np
import seaborn as sns
import pandas as pd

# Import PySwarms
import pyswarms as ps

# Some more magic so that the notebook will reload external python modules;
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
%matplotlib inline
```

Generating a toy dataset using scikit-learn

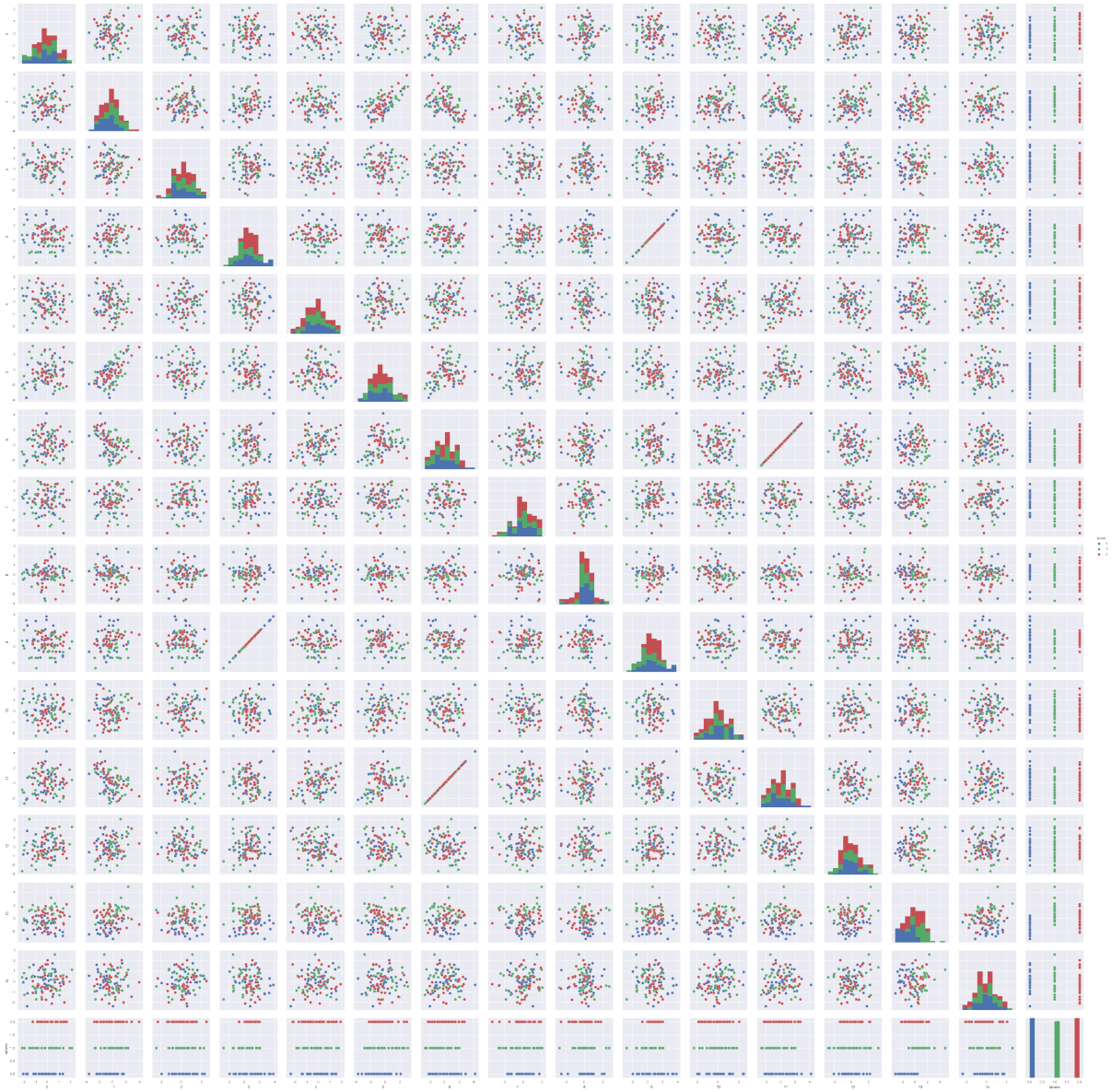
We'll be using `sklearn.datasets.make_classification` to generate a 100-sample, 15-dimensional dataset with three classes. We will then plot the distribution of the features in order to give us a qualitative assessment of the feature-space.

For our toy dataset, we will be rigging some parameters a bit. Out of the 15 features, we'll have only 4 that are informative, 1 that are redundant, and 2 that are repeated. Hopefully, we get to have Binary PSO select those that are informative, and prune those that are redundant or repeated.

```
from sklearn.datasets import make_classification
X, y = make_classification(n_samples=100, n_features=15, n_classes=3,
                           n_informative=4, n_redundant=1, n_repeated=2,
                           random_state=1)
```

```
# Plot toy dataset per feature
df = pd.DataFrame(X)
df['labels'] = pd.Series(y)

sns.pairplot(df, hue='labels');
```



As we can see, there are some features that causes the two classes to overlap with one another. These might be features that are better off unselected. On the other hand, we can see some feature combinations where the two classes are shown to be clearly separated. These features can hopefully be retained and selected by the binary PSO algorithm.

We will then use a simple logistic regression technique using `sklearn.linear_model.LogisticRegression` to perform classification. A simple test of accuracy will be used to assess the performance of the classifier.

Writing the custom-objective function

As seen above, we can write our objective function by simply taking the performance of the classifier (in this case, the accuracy), and the size of the feature subset divided by the total (that is, divided by 10), to return an error in the data. We'll now write our custom-objective function

```
from sklearn import linear_model

# Create an instance of the classifier
classifier = linear_model.LogisticRegression()

# Define objective function
def f_per_particle(m, alpha):
    """Computes for the objective function per particle

    Inputs
    ------
    m : numpy.ndarray
        Binary mask that can be obtained from BinaryPSO, will
        be used to mask features.
    alpha: float (default is 0.5)
        Constant weight for trading-off classifier performance
        and number of features

    Returns
    -------
    numpy.ndarray
        Computed objective function
    """
    total_features = 15
    # Get the subset of the features from the binary mask
    if np.count_nonzero(m) == 0:
        X_subset = X
    else:
        X_subset = X[:,m==1]
    # Perform classification and store performance in P
    classifier.fit(X_subset, y)
    P = (classifier.predict(X_subset) == y).mean()
    # Compute for the objective function
    j = (alpha * (1.0 - P)
        + (1.0 - alpha) * (1 - (X_subset.shape[1] / total_features)))

    return j
```

```
def f(x, alpha=0.88):
    """Higher-level method to do classification in the
    whole swarm.

    Inputs
    ------
    x: numpy.ndarray of shape (n_particles, dimensions)
        The swarm that will perform the search

    Returns
    -------
    numpy.ndarray of shape (n_particles, )
        The computed loss for each particle
    """
    n_particles = x.shape[0]
    j = [f_per_particle(x[i], alpha) for i in range(n_particles)]
    return np.array(j)
```

Using Binary PSO

With everything set-up, we can now use Binary PSO to perform feature selection. For now, we'll be doing a global-best solution by setting the number of neighbors equal to the number of particles. The hyperparameters are also set arbitrarily. Moreso, we'll also be setting the distance metric as 2 (truth is, it's not really relevant because each particle will see one another).

```
# Initialize swarm, arbitrary
options = {'c1': 0.5, 'c2': 0.5, 'w':0.9, 'k': 30, 'p':2}

# Call instance of PSO
dimensions = 15 # dimensions should be the number of features
optimizer.reset()
optimizer = ps.discrete.BinaryPSO(n_particles=30, dimensions=dimensions,␣
↪options=options)

# Perform optimization
cost, pos = optimizer.optimize(f, print_step=100, iters=1000, verbose=2)
```

```
Iteration 1/1000, cost: 0.2776
Iteration 101/1000, cost: 0.2792
Iteration 201/1000, cost: 0.2624
Iteration 301/1000, cost: 0.2632
Iteration 401/1000, cost: 0.2544
Iteration 501/1000, cost: 0.3208
Iteration 601/1000, cost: 0.2376
Iteration 701/1000, cost: 0.2944
Iteration 801/1000, cost: 0.3224
Iteration 901/1000, cost: 0.3464
================================
Optimization finished!
Final cost: 0.0000
Best value: 0.000000 1.000000 0.000000 1.000000 0.000000 1.000000 ...
```

We can then train the classifier using the positions found by running another instance of logistic regression. We can compare the performance when we're using the full set of features

```
# Create two instances of LogisticRegression
classfier = linear_model.LogisticRegression()

# Get the selected features from the final positions
X_selected_features = X[:,pos==1]  # subset

# Perform classification and store performance in P
classifier.fit(X_selected_features, y)

# Compute performance
subset_performance = (c1.predict(X_selected_features) == y).mean()


print('Subset performance: %.3f' % (subset_performance))
```

```
Subset performance: 0.680
```

Another important advantage that we have is that we were able to reduce the features (or do dimensionality reduction) on our data. This can save us from the curse of dimensionality, and may in fact speed up our classification.

Let's plot the feature subset that we have:

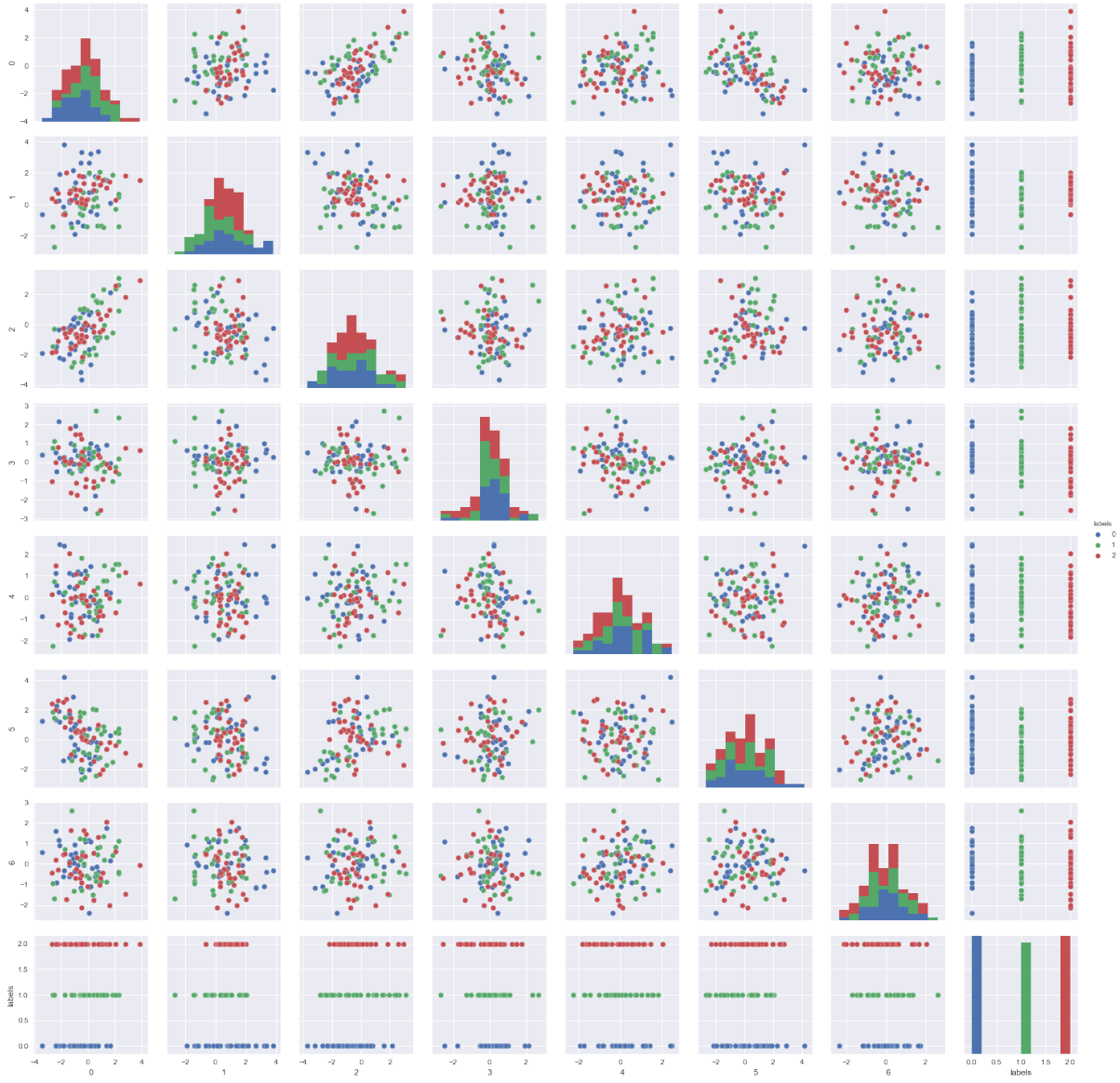
```
# Plot toy dataset per feature
df1 = pd.DataFrame(X_selected_features)
```

(continues on next page)

(continued from previous page)

```
df1['labels'] = pd.Series(y)

sns.pairplot(df1, hue='labels')
```



1.7.2 Solving the Inverse Kinematics problem using Particle Swarm Optimization

In this example, we are going to use the `pyswarms` library to solve a 6-DOF (Degrees of Freedom) Inverse Kinematics (IK) problem by treating it as an optimization problem. We will use the `pyswarms` library to find an *optimal* solution from a set of candidate solutions.

```
# Import modules
import numpy as np

# Import PySwarms
import pyswarms as ps

# Some more magic so that the notebook will reload external python modules;
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
```

(continues on next page)

(continued from previous page)

```
%load_ext autoreload
%autoreload 2
```

Introduction

Inverse Kinematics is one of the most challenging problems in robotics. The problem involves finding an optimal *pose* for a manipulator given the position of the end-tip effector as opposed to forward kinematics, where the end-tip position is sought given the pose or joint configuration. Normally, this position is expressed as a point in a coordinate system (e.g., in a Cartesian system with x, y and z coordinates). However, the pose of the manipulator can also be expressed as the collection of joint variables that describe the angle of bending or twist (in revolute joints) or length of extension (in prismatic joints).

IK is particularly difficult because an abundance of solutions can arise. Intuitively, one can imagine that a robotic arm can have multiple ways of reaching through a certain point. It's the same when you touch the table and move your arm without moving the point you're touching the table at. Moreover, the calculation of these positions can be very difficult. Simple solutions can be found for 3-DOF manipulators but trying to solve the problem for 6 or even more DOF can lead to challenging algebraic problems.

IK as an Optimization Problem

In this implementation, we are going to use a *6-DOF Stanford Manipulator* with 5 revolute joints and 1 prismatic joint. Furthermore, the constraints of the joints are going to be as follows:

Parameters	Lower Boundary	Upper Boundary
θ_1	$-\pi$	π
θ_2	$-\frac{\pi}{2}$	$\frac{\pi}{2}$
d_3	1	3
θ_4	$-\pi$	π
θ_5	$-\frac{5\pi}{36}$	$\frac{5\pi}{36}$
θ_6	$-\pi$	π

Table 1: *Physical constraints for the joint variables*

Now, if we are given an *end-tip position* (in this case a xyz coordinate) we need to find the optimal parameters with the constraints imposed in **Table 1**. These conditions are then sufficient in order to treat this problem as an optimization problem. We define our parameter vector $\mathbf{X}$ as follows:

$$\mathbf{X} := [\,\theta_1 \quad \theta_2 \quad d_3 \quad \theta_4 \quad \theta_5\,]$$

And for our end-tip position we define the target vector $\mathbf{T}$ as:

$$\mathbf{T} := [\,T_x \quad T_y \quad T_z\,]$$

We can then start implementing our optimization algorithm.

Initializing the Swarm

The main idea for PSO is that we set a swarm $\mathbf{S}$ composed of particles $\mathbf{P}_n$ into a search space in order to find the optimal solution. The movement of the swarm depends on the cognitive (c_1) and social (c_2) of all the particles. The cognitive component speaks of the particle's bias towards its personal best from its past experience (i.e., how attracted it is to its own best position). The social component controls how the particles are attracted to the best score found by the swarm (i.e., the global best). High c_1 paired with low c_2 values can often cause the swarm to stagnate. The inverse can cause the swarm to converge too fast, resulting in suboptimal solutions.

We define our particle $\mathbf{P}$ as:

$$\mathbf{P} := \mathbf{X}$$

And the swarm as being composed of N particles with certain positions at a timestep t:

$$\mathbf{S}_t := [\mathbf{P}_1 \quad \mathbf{P}_2 \quad ... \quad \mathbf{P}_N]$$

In this implementation, we designate $\mathbf{P}_1$ as the initial configuration of the manipulator at the zero-position. This means that the angles are equal to 0 and the link offset is also zero. We then generate the $N-1$ particles using a uniform distribution which is controlled by the hyperparameter ϵ.

Finding the global optimum

In order to find the global optimum, the swarm must be moved. This movement is then translated by an update of the current position given the swarm's velocity $\mathbf{V}$. That is:

$$\mathbf{S}_{t+1} = \mathbf{S}_t + \mathbf{V}_{t+1}$$

The velocity is then computed as follows:

$$\mathbf{V}_{t+1} = w\mathbf{V}_t + c_1 r_1(\mathbf{p}_{best} - \mathbf{p}) + c_2 r_2(\mathbf{g}_{best} - \mathbf{p})$$

Where r_1 and r_2 denote random values in the intervall $[0, 1]$, $\mathbf{p}_{best}$ is the best and $\mathbf{p}$ is the current personal position and $\mathbf{g}_{best}$ is the best position of all the particles. Moreover, w is the inertia weight that controls the "memory" of the swarm's previous position.

Preparations

Let us now see how this works with the `pyswarms` library. We use the point $[-2, 2, 3]$ as our target for which we want to find an optimal pose of the manipulator. We start by defining a function to get the distance from the current position to the target position:

```
def distance(query, target):
    x_dist = (target[0] - query[0])**2
    y_dist = (target[1] - query[1])**2
    z_dist = (target[2] - query[2])**2
    dist = np.sqrt(x_dist + y_dist + z_dist)
    return dist
```

We are going to use the distance function to compute the cost, the further away the more costly the position is.

The optimization algorithm needs some parameters (the swarm size, c_1, c_2 and ϵ). For the *options* (c_1,c_2 and w) we have to create a dictionary and for the constraints a tuple with a list of the respective minimal values and a list of the respective maximal values. The rest can be handled with variables. Additionally, we define the joint lengths to be 3 units long:

```
swarm_size = 20
dim = 6        # Dimension of X
epsilon = 1.0
options = {'c1': 1.5, 'c2':1.5, 'w':0.5}

constraints = (np.array([-np.pi , -np.pi/2 , 1 , -np.pi , -5*np.pi/36 , -np.pi]),
               np.array([np.pi  ,  np.pi/2 , 3 ,  np.pi ,  5*np.pi/36 ,  np.pi]))

d1 = d2 = d3 = d4 = d5 = d6 = 3
```

In order to obtain the current position, we need to calculate the matrices of rotation and translation for every joint. Here we use the Denvait-Hartenberg parameters for that. So we define a function that calculates these. The function uses the rotation angle and the extension d of a prismatic joint as input:

```
def getTransformMatrix(theta, d, a, alpha):
    T = np.array([[np.cos(theta) , -np.sin(theta)*np.cos(alpha) ,  np.
↪sin(theta)*np.sin(alpha) , a*np.cos(theta)],
                  [np.sin(theta) ,  np.cos(theta)*np.cos(alpha) , -np.
↪cos(theta)*np.sin(alpha) , a*np.sin(theta)],
                  [0             ,  np.sin(alpha)               ,  np.cos(alpha)
↪            , d              ],
                  [0             ,  0                           ,  0
↪            , 1              ]
                   ])
    return T
```

Now we can calculate the transformation matrix to obtain the end tip position. For this we create another function that takes our vector $\mathbf{X}$ with the joint variables as input:

```
def get_end_tip_position(params):
    # Create the transformation matrices for the respective joints
    t_00 = np.array([[1,0,0,0],[0,1,0,0],[0,0,1,0],[0,0,0,1]])
    t_01 = getTransformMatrix(params[0] , d2        , 0 , -np.pi/2)
    t_12 = getTransformMatrix(params[1] , d2        , 0 , -np.pi/2)
    t_23 = getTransformMatrix(0         , params[2] , 0 , -np.pi/2)
    t_34 = getTransformMatrix(params[3] , d4        , 0 , -np.pi/2)
    t_45 = getTransformMatrix(params[4] , 0         , 0 ,  np.pi/2)
    t_56 = getTransformMatrix(params[5] , d6        ,0  ,  0)

    # Get the overall transformation matrix
    end_tip_m = t_00.dot(t_01).dot(t_12).dot(t_23).dot(t_34).dot(t_45).dot(t_56)

    # The coordinates of the end tip are the 3 upper entries in the 4th column
    pos = np.array([end_tip_m[0,3],end_tip_m[1,3],end_tip_m[2,3]])
    return pos
```

The last thing we need to prepare in order to run the algorithm is the actual function that we want to optimize. We just need to calculate the distance between the position of each swarm particle and the target point:

```
def opt_func(X):
    n_particles = X.shape[0]  # number of particles
    target = np.array([-2,2,3])
    dist = [distance(get_end_tip_position(X[i]), target) for i in range(n_
↪particles)]
    return np.array(dist)
```

Running the algorithm

Braced with these preparations we can finally start using the algorithm:

```
%%time
# Call an instance of PSO
optimizer = ps.single.GlobalBestPSO(n_particles=swarm_size,
                                    dimensions=dim,
                                    options=options,
                                    bounds=constraints)

# Perform optimization
cost, joint_vars = optimizer.optimize(opt_func, iters=1000)
```

```
INFO:pyswarms.single.global_best:Iteration 1/1000, cost: 0.9638223076369133
INFO:pyswarms.single.global_best:Iteration 101/1000, cost: 2.5258875519324167e-07
INFO:pyswarms.single.global_best:Iteration 201/1000, cost: 4.7236564972673785e-14
INFO:pyswarms.single.global_best:Iteration 301/1000, cost: 0.0
```

(continues on next page)

(continued from previous page)

```
INFO:pyswarms.single.global_best:Iteration 401/1000, cost: 0.0
INFO:pyswarms.single.global_best:Iteration 501/1000, cost: 0.0
INFO:pyswarms.single.global_best:Iteration 601/1000, cost: 0.0
INFO:pyswarms.single.global_best:Iteration 701/1000, cost: 0.0
INFO:pyswarms.single.global_best:Iteration 801/1000, cost: 0.0
INFO:pyswarms.single.global_best:Iteration 901/1000, cost: 0.0
INFO:pyswarms.single.global_best:================================
Optimization finished!
Final cost: 0.0000
Best value: [ -2.182725 1.323111 1.579636 ...]
```

Now let's see if the algorithm really worked and test the output for `joint_vars`:

```
print(get_end_tip_position(joint_vars))
```

```
[-2.  2.  3.]
```

Hooray! That's exactly the position we wanted the tip to be in. Of course this example is quite primitive. Some extensions of this idea could involve the consideration of the current position of the manipulator and the amount of rotation and extension in the optimization function such that the result is the path with the least movement.

1.8 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

1.8.1 Types of Contributions

Report Bugs

Report bugs at https://github.com/ljvmiranda921/pyswarms/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it. Those that are tagged with "first-timers-only" is suitable for those getting started in open-source software.

Write Documentation

PySwarms could always use more documentation, whether as part of the official PySwarms docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/ljvmiranda921/pyswarms/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

1.8.2 Get Started!

Ready to contribute? Here's how to set up *pyswarms* for local development.

1. Fork the *pyswarms* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/pyswarms.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv pyswarms
$ cd pyswarms/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox. In addition, ensure that your code is formatted using black:

```
$ flake8 pyswarms tests
$ black pyswarms tests
$ python setup.py test or py.test
$ tox
```

   To get flake8, black, and tox, just pip install them into your virtualenv. If you wish, you can add pre-commit hooks for both flake8 and black to make all formatting easier.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

   In brief, commit messages should follow these conventions:

   - Always contain a subject line which briefly describes the changes made. For example "Update CONTRIBUTING.rst".
   - Subject lines should not exceed 50 characters.

- The commit body should contain context about the change - how the code worked before, how it works now and why you decided to solve the issue in the way you did.

More detail on commit guidelines can be found at https://chris.beams.io/posts/git-commit

7. Submit a pull request through the GitHub website.

1.8.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

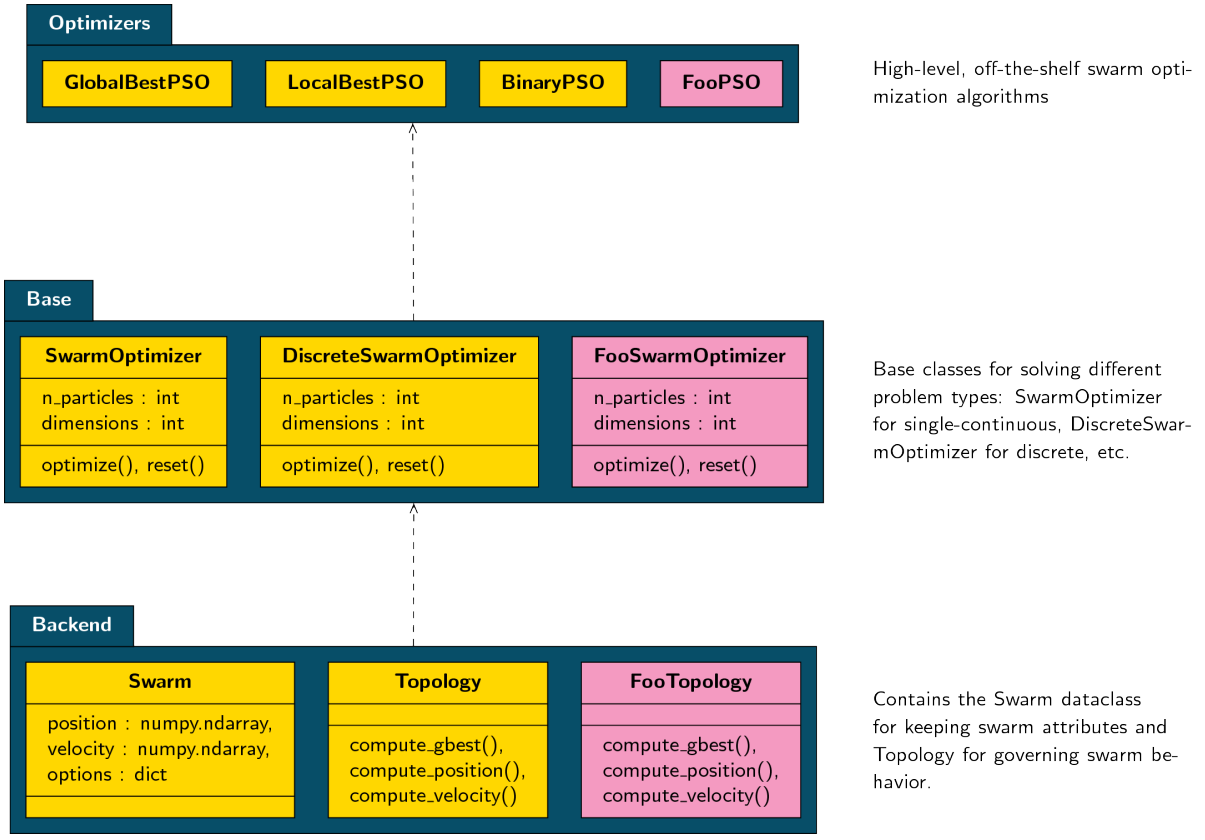
1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.7, 3.4, 3.5, and above. Check https://travis-ci.org/ljvmiranda921/pyswarms/pull_requests and make sure that the tests pass for all supported Python versions.

1.9 Understanding the PySwarms API

There are three main layers in PySwarms' main API:

- **Optimizers**: includes all off-the-shelf implementations of most swarm intelligence algorithms
- **Base**: base API where most Optimizer implementations were based upon. Each Base module is designed with respect to the problem domain they're trying to solve: single-continuous, discrete, *(in the future)* multiobjective, constrained, etc.
- **Backend**: backend API that exposes common operations for any swarm algorithm such as swarm initialization, global best computation, nearest neighbor search, etc.

You can find the structure of the main PySwarms API in the figure below:



When contributing to PySwarms, you can start off with any of the Layers specified above. Right now, we would really appreciate contributions from the Base Layer below. Some of which that need some dedicated contributions:

- ConstrainedOptimizer (in Base Layer)
- MultiObjectiveOptimizer (in Base Layer)
- Different Topologies (in Backend Layer)

If we can have a strong set of native APIs for the low-level layers, it will then be very easy to implement different swarm algorithms. Of course, for your personal needs, you can simply inherit any of the classes in PySwarms and modify them according to your own specifications.

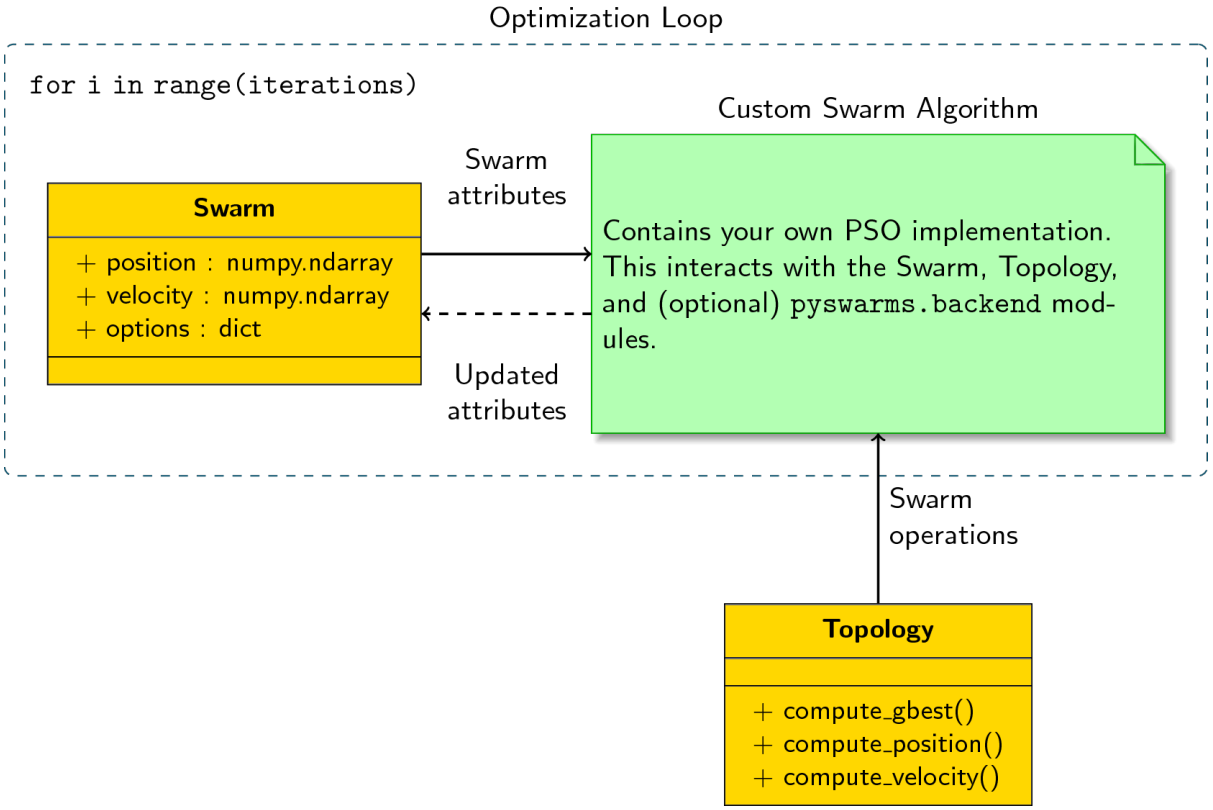
Remember, when you want to implement your own Optimizer, there is no need to go from Backend to Optimizers layer. Instead, you can just import the *pyswarms.backend.swarms.Swarm* class and the classes in the `pyswarms.backend.topology` module.

1.10 Writing your own optimization loop

The backend module provides a lot of helper methods for you to customize your swarm implementation. This gives you a black-box approach by requiring you to write your own optimization-loop.

There are two important components for any swarm implementation:

- The **Swarm** class, containing all important attributes and properties of the swarm; and
- The **Topology** class, governing how the swarm will behave during optimization.



The main idea is that for every iteration, you interact with the Swarm class using the methods found in the Topology class (or optionally, in *pyswarms.backend.operators*). You continuously take the attributes present in Swarm, and update them using the operations your algorithm requires. Together with some methods found in *pyswarms.backend.generators* and *pyswarms.backend.operators*, it is possible to create different kinds of swarm implementations.

1.10.1 The Swarm Class

pyswarms.backend.swarms.Swarm acts as a data-class that keeps all necessary attributes in a given swarm implementation. You initialize it by providing the initial position and velocity matrices. For the current iteration, you can obtain the following information from the class:

- `position`: the current position-matrix of the swarm. Each row is a particle and each column is its position on a given dimension.
- `velocity`: the current velocity-matrix of the swarm. Each row is a particle and each column is its velocity on a given dimension.
- `pbest_pos`: the personal best position of each particle that corresponds to the personal best cost.
- `pbest_cost`: the personal best fitness attained by the particle since the first iteration.
- `best_pos`: the best position found by the swarm that corresponds to the best cost.
- `best_cost`: the best fitness found by the swarm.
- `options`: additional options that you can use for your particular needs. As an example, the `GlobalBestPSO` implementation uses this to store the cognitive and social parameters of the swarm.

1.10.2 The Topology Class

`pyswarms.backend.base.topology` houses all operations that you can use on the Swarm attributes. Currently, the Star and Ring topologies are implemented, but more topologies will still be done in the future. A Topology implements three methods governing swarm behavior:

- `compute_gbest`: computes the best particle (both cost and position) given a swarm instance.
- `compute_position`: computes the next position of the swarm given its current position.
- `compute_velocity`: computes the velocity of the swarm given its attributes.

Needless to say, these three methods will differ depending on the topology present. All these methods take in an instance of the `Swarm` class, and outputs the necessary matrices. The advantage of using this class is that it abstracts away all the internals of implementing a swarm algorithm. You just need to provide the topology, and call its methods right away.

1.11 Contributing your own optimizer

PySwarms aims to be the go-to library for various PSO implementations, so if you are a researcher in swarm intelligence or a developer who wants to contribute, then read on this guide!

As a preliminary, here is a checklist whenever you will implement an optimizer:

- Propose an optimizer
- Write optimizer by inheriting from base classes
- Write a unit test

1.11.1 Proposing an optimizer

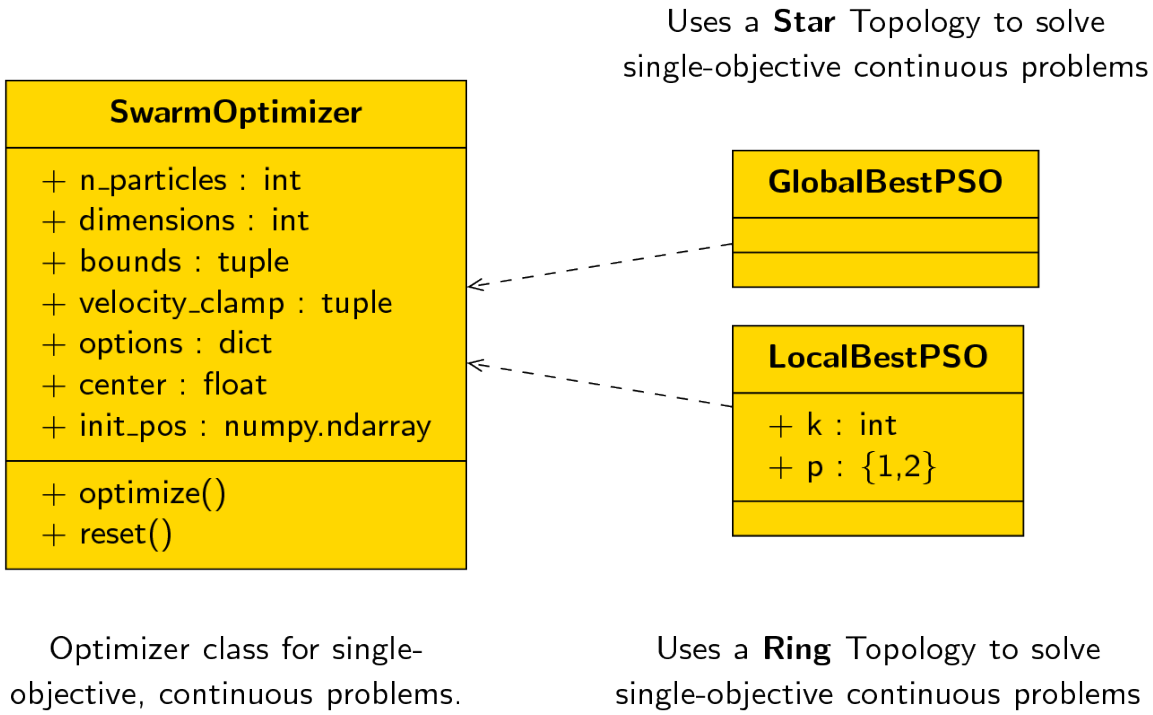
We wanted to make sure that PySwarms is highly-usable, and thus it is important that optimizers included in this library are either (1) classic textbook-PSO techniques or (2) highly-cited, published, optimization algorithms.

In case you wanted to include your optimization algorithm in this library, please raise an issue and add a short abstract on what your optimizer does. A link to a published paper (it's okay if it's behind a paywall) would be really helpful!

1.11.2 Inheriting from base classes

Most optimizers in this library inherit its attributes and methods from a set of built-in base classes. You can check the existing classes in *pyswarms.base*.

For example, if we take the *pyswarms.base.base_single* class, a base-class for standard single-objective continuous optimization algorithms such as global-best PSO (*pyswarms.single.global_best*) and local-best PSO (*pyswarms.single.local_best*), we can see that it inherits a set of methods as seen below:



The required methods can be seen in the base classes, and will raise a `NotImplementedError` if not called. Additional methods, private or not, can also be added depending on the needs of your optimizer.

A short note on keyword arguments

The role of keyword arguments, or kwargs in short, is to act as a container for all other parameters needed for the optimizer. You can define these things in your code, and create assertions to make all of them required. However, note that in some implementations, required `options` might include `c1`, `c2`, and `w`. This is the case in `pyswarms.base.bases` for instance.

A short note on `assertions()`

You might notice that in most base classes, an `assertions()` method is being called. This aims to check if the user-facing input are correct. Although the method is called "assertions", please make all user-facing catches as raised Exceptions.

A short note on `__init__.py`

We make sure that everything can be imported when the whole `pyswarms` library is called. Thus, please make sure to also edit the accompanying `__init__.py` file in the directory you are working on.

For example, if you write your optimizer class `MyOptimizer` inside a file called `my_optimizer.py`, and you are working under the `/single` directory, please update the `__init__.py` like the following:

```
from .global_best import GlobalBestPSO
from .local_best import LocalBestPSO
# Add your module
from .my_optimizer import MyOptimizer

__all__ = [
    "GlobalBestPSO",
    "LocalBestPSO",
    "MyOptimizer" # Add your class
    ]
```

This ensures that it will be automatically initialized when the whole library is imported.

1.11.3 Writing unit tests

Testing is an important element of developing PySwarms and we want everything to be as smooth as possible. Especially, when working on the build and integrating new features. In this case, we provide the `tests` module in the package. For writing the test, we use the `pytest` module. In case you add a test for your optimizer, use the same naming conventions that were used in the existing ones.

You can perform separate checks by

```
$ python -m pytest tests.optimizers.<test_myoptimizer>
```

For more details on running the tests see here.

1.12 Backend

The main workhorse of PySwarms is the backend module. It contains various primitive methods and classes to help you create your own custom swarm implementation. The high-level PSO implementations in this library such as GlobalBestPSO and LocalBestPSO were built using the backend module.

1.12.1 pyswarms.backend package

You can import all the native helper methods in this package using the command:

```
import pyswarms.backend as P
```

Then call the methods found in each module. Note that these methods interface with the Swarm class provided in the `pyswarms.backend.swarms` module.

pyswarms.backend.generators module

Swarm Generation Backend

This module abstracts how a swarm is generated. You can see its implementation in our base classes. In addition, you can use all the methods here to dictate how a swarm is initialized for your custom PSO.

pyswarms.backend.generators.**create_swarm**(*n_particles*, *dimensions*, *discrete=False*, *binary=False*, *options={}*, *bounds=None*, *center=1.0*, *init_pos=None*, *clamp=None*)

Abstract the generate_swarm() and generate_velocity() methods

Parameters

- **n_particles** (*int*) – number of particles to be generated in the swarm.
- **dimensions** (*int*) – number of dimensions to be generated in the swarm

- **options** (dict (default is empty dict `{}`)) – Swarm options, for example, c1, c2, etc.
- **discrete** (bool (default is `False`)) – Creates a discrete swarm
- **binary** (bool (default is `False`)) – generate a binary matrix
- **bounds** (tuple of `np.ndarray` or list (default is `None`)) – a tuple of size 2 where the first entry is the minimum bound while the second entry is the maximum bound. Each array must be of shape `(dimensions,)`.
- **center** (`numpy.ndarray` (default is `1`)) – a list of initial positions for generating the swarm
- **init_pos** (`numpy.ndarray` (default is `None`)) – option to explicitly set the particles' initial positions. Set to `None` if you wish to generate the particles randomly.
- **clamp** (tuple of floats (default is `None`)) – a tuple of size 2 where the first entry is the minimum velocity and the second entry is the maximum velocity. It sets the limits for velocity clamping.

Returns a Swarm class

Return type *pyswarms.backend.swarms.Swarm*

pyswarms.backend.generators.**generate_discrete_swarm**(*n_particles*, *dimensions*, *binary=False*, *init_pos=None*)

Generate a discrete swarm

Parameters

- **n_particles** (*int*) – number of particles to be generated in the swarm.
- **dimensions** (*int*) – number of dimensions to be generated in the swarm.
- **binary** (bool (default is `False`)) – generate a binary matrix
- **init_pos** (`numpy.ndarray` (default is `None`)) – option to explicitly set the particles' initial positions. Set to `None` if you wish to generate the particles randomly.

Returns swarm matrix of shape (n_particles, n_dimensions)

Return type numpy.ndarray

Raises

- `ValueError` – When init_pos during binary=True does not contain two unique values.
- `TypeError` – When the argument passed to n_particles or dimensions is incorrect.

pyswarms.backend.generators.**generate_swarm**(*n_particles*, *dimensions*, *bounds=None*, *center=1.0*, *init_pos=None*)

Generate a swarm

Parameters

- **n_particles** (*int*) – number of particles to be generated in the swarm.
- **dimensions** (*int*) – number of dimensions to be generated in the swarm
- **bounds** (tuple of `np.ndarray` or list (default is `None`)) – a tuple of size 2 where the first entry is the minimum bound while the second entry is the maximum bound. Each array must be of shape `(dimensions,)`.
- **center** (`numpy.ndarray` or `float` (default is `1`)) – controls the mean or center whenever the swarm is generated randomly.
- **init_pos** (`numpy.ndarray` (default is `None`)) – option to explicitly set the particles' initial positions. Set to `None` if you wish to generate the particles randomly.

Returns swarm matrix of shape (n_particles, n_dimensions)

Return type numpy.ndarray

Raises

- `ValueError` – When the shapes and values of bounds, dimensions, and init_pos are inconsistent.
- `TypeError` – When the argument passed to bounds is not an iterable.

`pyswarms.backend.generators.`**`generate_velocity`**(*n_particles*, *dimensions*, *clamp=None*)

Initialize a velocity vector

Parameters

- **`n_particles`** (*int*) – number of particles to be generated in the swarm.
- **`dimensions`** (*int*) – number of dimensions to be generated in the swarm.
- **`clamp`** (tuple of floats (default is `None`)) – a tuple of size 2 where the first entry is the minimum velocity and the second entry is the maximum velocity. It sets the limits for velocity clamping.

Returns velocity matrix of shape (n_particles, dimensions)

Return type numpy.ndarray

pyswarms.backend.handlers module

Handlers

This module provides Handler classes for the position as well as the velocity of particles. This is necessary when boundary conditions are imposed on the PSO algorithm. Particles that do not stay inside these boundary conditions have to be handled by either adjusting their position after they left the bounded search space or adjusting their velocity when it would position them outside the search space. In particular, this approach is important if the optimium of a function is near the boundaries. For the following documentation let $x_{i,t,d}$ be the d th coordinate of the particle i 's position vector at the time t, lb the vector of the lower boundaries and ub the vector of the upper boundaries. The algorithms in this module are adapted from [SH2010]

[SH2010] Sabine Helwig, "Particle Swarms for Constrained Optimization", PhD thesis, Friedrich-Alexander Universität Erlangen-Nürnberg, 2010.

`class` `pyswarms.backend.handlers.`**`BoundaryHandler`**(*strategy*)

`intermediate`(*position*, *bounds*, ***kwargs*)

Set the particle to an intermediate position

This method resets particles that exceed the bounds to an intermediate position between the boundary and their earlier position. Namely, it changes the coordinate of the out-of-bounds axis to the middle value between the previous position and the boundary of the axis. The following equation describes this strategy:

$$x_{i,t,d} = \begin{cases} \frac{1}{2}\left(x_{i,t-1,d} + lb_d\right) & \text{if } x_{i,t,d} < lb_d \\ \frac{1}{2}\left(x_{i,t-1,d} + ub_d\right) & \text{if } x_{i,t,d} > ub_d \\ x_{i,t,d} & \text{otherwise} \end{cases}$$

`nearest`(*position*, *bounds*, ***kwargs*)

Set position to nearest bound

This method resets particles that exceed the bounds to the nearest available boundary. For every axis on which the coordiantes of the particle surpasses the boundary conditions the coordinate is set to the

respective bound that it surpasses. The following equation describes this strategy:

$$x_{i,t,d} = \begin{cases} lb_d & \text{if } x_{i,t,d} < lb_d \\ ub_d & \text{if } x_{i,t,d} > ub_d \\ x_{i,t,d} & \text{otherwise} \end{cases}$$

periodic(*position*, *bounds*, ***kwargs*)
Sets the particles a periodic fashion

This method resets the particles that exeed the bounds by using the modulo function to cut down the position. This creates a virtual, periodic plane which is tiled with the search space. The following equation describtes this strategy:

$$x_{i,t,d} = \begin{cases} ub_d - (lb_d - x_{i,t,d}) \mod s_d & \text{if } x_{i,t,d} < lb_d \\ lb_d + (x_{i,t,d} - ub_d) \mod s_d & \text{if } x_{i,t,d} > ub_d \\ x_{i,t,d} & \text{otherwise} \end{cases}$$

with

$$s_d = |ub_d - lb_d|$$

random(*position*, *bounds*, ***kwargs*)
Set position to random location

This method resets particles that exeed the bounds to a random position inside the boundary conditions.

reflective(*position*, *bounds*, ***kwargs*)
Reflect the particle at the boundary

This method reflects the particles that exceed the bounds at the respective boundary. This means that the amount that the component which is orthogonal to the exceeds the boundary is mirrored at the boundary. The reflection is repeated until the position of the particle is within the boundaries. The following algorithm describes the behaviour of this strategy:

$$\text{while } x_{i,t,d} \notin [lb_d,\, ub_d]$$
$$\text{do the following:}$$

$$x_{i,t,d} = \begin{cases} 2 \cdot lb_d - x_{i,t,d} & \text{if } x_{i,t,d} < lb_d \\ 2 \cdot ub_d - x_{i,t,d} & \text{if } x_{i,t,d} > ub_d \\ x_{i,t,d} & \text{otherwise} \end{cases}$$

shrink(*position*, *bounds*, ***kwargs*)
Set the particle to the boundary

This method resets particles that exceed the bounds to the intersection of its previous velocity and the boundary. This can be imagined as shrinking the previous velocity until the particle is back in the valid search space. Let $\sigma_{i,t,d}$ be the d th shrinking value of the i th particle at the time t and $v_{i,t}$ the velocity

of the i th particle at the time t. Then the new position is computed by the following equation:

$$\mathbf{x}_{i,t} = \mathbf{x}_{i,t-1} + \sigma_{i,t}\mathbf{v}_{i,t}$$

with

$$\sigma_{i,t,d} = \begin{cases} \frac{lb_d - x_{i,t-1,d}}{v_{i,t,d}} & \text{if } x_{i,t,d} < lb_d \\ \frac{ub_d - x_{i,t-1,d}}{v_{i,t,d}} & \text{if } x_{i,t,d} > ub_d \\ 1 & \text{otherwise} \end{cases}$$

and

$$\sigma_{i,t} = \min_{d=1\ldots n} \sigma_{i,t,d}$$

class pyswarms.backend.handlers.**HandlerMixin**
A HandlerMixing class

This class offers some basic functionality for the Handlers.

class pyswarms.backend.handlers.**VelocityHandler**(*strategy*)

adjust(*velocity*, *clamp=None*, ***kwargs*)
Adjust the velocity to the new position

The velocity is adjusted such that the following equation holds: .. math:

```
\mathbf{v_{i,t}} = \mathbf{x_{i,t}} - \mathbf{x_{i,t-1}}
```

Note: This method should only be used in combination with a position handling operation.

invert(*velocity*, *clamp=None*, ***kwargs*)
Invert the velocity if the particle is out of bounds

The velocity is inverted and shrinked. The shrinking is determined by the kwarg `z`. The default shrinking factor is `0.5`. For all velocities whose particles are out of bounds the following equation is applied: .. math:

```
\mathbf{v_{i,t}} = -z\mathbf{v_{i,t}}
```

unmodified(*velocity*, *clamp=None*, ***kwargs*)
Leaves the velocity unchanged

zero(*velocity*, *clamp=None*, ***kwargs*)
Set velocity to zero if the particle is out of bounds

pyswarms.backend.operators module

Swarm Operation Backend

This module abstracts various operations in the swarm such as updating the personal best, finding neighbors, etc. You can use these methods to specify how the swarm will behave.

pyswarms.backend.operators.**compute_objective_function**(*swarm*, *objective_func*, *pool=None*, ***kwargs*)
Evaluate particles using the objective function

This method evaluates each particle in the swarm according to the objective function passed.

If a pool is passed, then the evaluation of the particles is done in parallel using multiple processes.

Parameters

- **swarm** (pyswarms.backend.swarms.Swarm) – a Swarm instance
- **objective_func** (*function*) – objective function to be evaluated
- **pool** (*multiprocessing.Pool*) – multiprocessing.Pool to be used for parallel particle evaluation
- **kwargs** (*dict*) – arguments for the objective function

Returns Cost-matrix for the given swarm

Return type numpy.ndarray

pyswarms.backend.operators.**compute_pbest**(*swarm*)

Update the personal best score of a swarm instance

You can use this method to update your personal best positions.

```
import pyswarms.backend as P
from pyswarms.backend.swarms import Swarm

my_swarm = P.create_swarm(n_particles, dimensions)

# Inside the for-loop...
for i in range(iters):
    # It updates the swarm internally
    my_swarm.pbest_pos, my_swarm.pbest_cost = P.update_pbest(my_swarm)
```

It updates your `current_pbest` with the personal bests acquired by comparing the (1) cost of the current positions and the (2) personal bests your swarm has attained.

If the cost of the current position is less than the cost of the personal best, then the current position replaces the previous personal best position.

Parameters **swarm** (*pyswarms.backend.swarm.Swarm*) – a Swarm instance

Returns

- *numpy.ndarray* – New personal best positions of shape `(n_particles, n_dimensions)`
- *numpy.ndarray* – New personal best costs of shape `(n_particles,)`

pyswarms.backend.operators.**compute_position**(*swarm*, *bounds*, *bh*)

Update the position matrix

This method updates the position matrix given the current position and the velocity. If bounded, the positions are handled by a `BoundaryHandler` instance.

```
import pyswarms.backend as P
from pyswarms.swarms.backend import Swarm, VelocityHandler

my_swarm = P.create_swarm(n_particles, dimensions)
my_bh = BoundaryHandler(strategy="intermediate")

for i in range(iters):
    # Inside the for-loop
    my_swarm.position = compute_position(my_swarm, bounds, my_bh)
```

Parameters

- **swarm** (pyswarms.backend.swarms.Swarm) – a Swarm instance

- **bounds** (tuple of `np.ndarray` or list (default is `None`)) – a tuple of size 2 where the first entry is the minimum bound while the second entry is the maximum bound. Each array must be of shape `(dimensions,)`.
- **bh** (`pyswarms.backend.handlers.BoundaryHandler`) – a BoundaryHandler object with a specified handling strategy For further information see *pyswarms.backend.handlers*.

Returns New position-matrix

Return type numpy.ndarray

`pyswarms.backend.operators.`**`compute_velocity`**(*swarm*, *clamp*, *vh*, *bounds=None*)

Update the velocity matrix

This method updates the velocity matrix using the best and current positions of the swarm. The velocity matrix is computed using the cognitive and social terms of the swarm. The velocity is handled by a `VelocityHandler`.

A sample usage can be seen with the following:

```
import pyswarms.backend as P
from pyswarms.swarms.backend import Swarm, VelocityHandler

my_swarm = P.create_swarm(n_particles, dimensions)
my_vh = VelocityHandler(strategy="invert")

for i in range(iters):
    # Inside the for-loop
    my_swarm.velocity = compute_velocity(my_swarm, clamp, my_vh, bounds)
```

Parameters

- **swarm** (`pyswarms.backend.swarms.Swarm`) – a Swarm instance
- **clamp** (tuple of floats (default is `None`)) – a tuple of size 2 where the first entry is the minimum velocity and the second entry is the maximum velocity. It sets the limits for velocity clamping.
- **vh** (`pyswarms.backend.handlers.VelocityHandler`) – a VelocityHandler object with a specified handling strategy. For further information see *pyswarms.backend.handlers*.
- **bounds** (tuple of `np.ndarray` or list (default is `None`)) – a tuple of size 2 where the first entry is the minimum bound while the second entry is the maximum bound. Each array must be of shape `(dimensions,)`.

Returns Updated velocity matrix

Return type numpy.ndarray

1.12.2 pyswarms.handlers package

This package implements different handling strategies for the optimiziation using boundary conditions. The strategies help avoiding that particles leave the defined search space. There are two `Handler` classes that provide these functionalities, the `BoundaryHandler` and the `VelocityHandler`.

pyswarms.handlers class

Handlers

This module provides Handler classes for the position as well as the velocity of particles. This is necessary when boundary conditions are imposed on the PSO algorithm. Particles that do not stay inside these boundary

conditions have to be handled by either adjusting their position after they left the bounded search space or adjusting their velocity when it would position them outside the search space. In particular, this approach is important if the optimium of a function is near the boundaries. For the following documentation let $x_{i,t,d}$ be the d th coordinate of the particle i 's position vector at the time t, lb the vector of the lower boundaries and ub the vector of the upper boundaries. The algorithms in this module are adapted from [SH2010]

[SH2010] Sabine Helwig, "Particle Swarms for Constrained Optimization", PhD thesis, Friedrich-Alexander Universität Erlangen-Nürnberg, 2010.

class pyswarms.backend.handlers.**BoundaryHandler**(*strategy*)

Bases: *pyswarms.backend.handlers.HandlerMixin*

__call__(*position*, *bounds*, ***kwargs*)

Apply the selected strategy to the position-matrix given the bounds

Parameters

- **position** (*np.ndarray*) – The swarm position to be handled
- **bounds** (tuple of np.ndarray or list) – a tuple of size 2 where the first entry is the minimum bound while the second entry is the maximum bound. Each array must be of shape (dimensions,)
- **kwargs** (*dict*) –

Returns the adjusted positions of the swarm

Return type numpy.ndarray

__init__(*strategy*)

A BoundaryHandler class

This class offers a way to handle boundary conditions. It contains methods to repair particle positions outside of the defined boundaries. Following strategies are available for the handling:

- **Nearest:** Reposition the particle to the nearest bound.
- **Random:** Reposition the particle randomly in between the bounds.
- **Shrink:** Shrink the velocity of the particle such that it lands on the bounds.
- **Reflective:** Mirror the particle position from outside the bounds to inside the bounds.
- **Intermediate:** Reposition the particle to the midpoint between its current position on the bound surpassing axis and the bound itself. This only adjusts the axes that surpass the boundaries.

The BoundaryHandler can be called as a function to use the strategy that is passed at initialization to repair boundary issues. An example for the usage:

```
from pyswarms.backend import operators as op
from pyswarms.backend.handlers import BoundaryHandler

bh = BoundaryHandler(strategy="reflective")
ops.compute_position(swarm, bounds, handler=bh)
```

By passing the handler, the compute_position() function now has the ability to reset the particles by calling the BoundaryHandler inside.

strategy

str – The strategy to use. To see all available strategies, call BoundaryHandler.strategies

intermediate(*position*, *bounds*, ***kwargs*)

Set the particle to an intermediate position

This method resets particles that exceed the bounds to an intermediate position between the boundary and their earlier position. Namely, it changes the coordinate of the out-of-bounds axis to the middle

value between the previous position and the boundary of the axis. The following equation describes this strategy:

$$x_{i,t,d} = \begin{cases} \frac{1}{2}\left(x_{i,t-1,d} + lb_d\right) & \text{if } x_{i,t,d} < lb_d \\ \frac{1}{2}\left(x_{i,t-1,d} + ub_d\right) & \text{if } x_{i,t,d} > ub_d \\ x_{i,t,d} & \text{otherwise} \end{cases}$$

nearest (*position*, *bounds*, ***kwargs*)
Set position to nearest bound

This method resets particles that exceed the bounds to the nearest available boundary. For every axis on which the coordiantes of the particle surpasses the boundary conditions the coordinate is set to the respective bound that it surpasses. The following equation describes this strategy:

$$x_{i,t,d} = \begin{cases} lb_d & \text{if } x_{i,t,d} < lb_d \\ ub_d & \text{if } x_{i,t,d} > ub_d \\ x_{i,t,d} & \text{otherwise} \end{cases}$$

periodic (*position*, *bounds*, ***kwargs*)
Sets the particles a periodic fashion

This method resets the particles that exeed the bounds by using the modulo function to cut down the position. This creates a virtual, periodic plane which is tiled with the search space. The following equation describtes this strategy:

$$x_{i,t,d} = \begin{cases} ub_d - (lb_d - x_{i,t,d}) \mod s_d & \text{if } x_{i,t,d} < lb_d \\ lb_d + (x_{i,t,d} - ub_d) \mod s_d & \text{if } x_{i,t,d} > ub_d \\ x_{i,t,d} & \text{otherwise} \end{cases}$$

with

$$s_d = |ub_d - lb_d|$$

random (*position*, *bounds*, ***kwargs*)
Set position to random location

This method resets particles that exeed the bounds to a random position inside the boundary conditions.

reflective (*position*, *bounds*, ***kwargs*)
Reflect the particle at the boundary

This method reflects the particles that exceed the bounds at the respective boundary. This means that the amount that the component which is orthogonal to the exceeds the boundary is mirrored at the boundary. The reflection is repeated until the position of the particle is within the boundaries. The following algorithm describes the behaviour of this strategy:

$$\text{while } x_{i,t,d} \notin [lb_d,\, ub_d]$$
$$\text{do the following:}$$

$$x_{i,t,d} = \begin{cases} 2 \cdot lb_d - x_{i,t,d} & \text{if } x_{i,t,d} < lb_d \\ 2 \cdot ub_d - x_{i,t,d} & \text{if } x_{i,t,d} > ub_d \\ x_{i,t,d} & \text{otherwise} \end{cases}$$

shrink (*position*, *bounds*, ***kwargs*)
Set the particle to the boundary

This method resets particles that exceed the bounds to the intersection of its previous velocity and the boundary. This can be imagined as shrinking the previous velocity until the particle is back in the valid

search space. Let $\sigma_{i,t,d}$ be the d th shrinking value of the i th particle at the time t and $v_{i,t}$ the velocity of the i th particle at the time t. Then the new position is computed by the following equation:

$$\mathbf{x}_{i,t} = \mathbf{x}_{i,t-1} + \sigma_{i,t}\mathbf{v}_{i,t}$$

with

$$\sigma_{i,t,d} = \begin{cases} \frac{lb_d - x_{i,t-1,d}}{v_{i,t,d}} & \text{if } x_{i,t,d} < lb_d \\ \frac{ub_d - x_{i,t-1,d}}{v_{i,t,d}} & \text{if } x_{i,t,d} > ub_d \\ 1 & \text{otherwise} \end{cases}$$

and

$$\sigma_{i,t} = \min_{d=1\ldots n} \sigma_{i,t,d}$$

class pyswarms.backend.handlers.**HandlerMixin**
Bases: object

A HandlerMixing class

This class offers some basic functionality for the Handlers.

class pyswarms.backend.handlers.**VelocityHandler**(*strategy*)
Bases: *pyswarms.backend.handlers.HandlerMixin*

__call__(*velocity*, *clamp*, ***kwargs*)
Apply the selected strategy to the velocity-matrix given the bounds

Parameters

- **velocity** (*np.ndarray*) – The swarm position to be handled
- **clamp** (tuple of np.ndarray or list) – a tuple of size 2 where the first entry is the minimum clamp while the second entry is the maximum clamp. Each array must be of shape (dimensions,)
- **kwargs** (*dict*) –

Returns the adjusted positions of the swarm

Return type numpy.ndarray

__init__(*strategy*)
A VelocityHandler class

This class offers a way to handle velocities. It contains methods to repair the velocities of particles that exceeded the defined boundaries. Following strategies are available for the handling:

- **Unmodified:** Returns the unmodified velocites.
- **Adjust** Returns the velocity that is adjusted to be the distance between the current and the previous position.
- **Invert** Inverts and shrinks the velocity by the factor -z.
- **Zero** Sets the velocity of out-of-bounds particles to zero.

adjust(*velocity*, *clamp=None*, ***kwargs*)
Adjust the velocity to the new position

The velocity is adjusted such that the following equation holds: .. math:

```
\mathbf{v_{i,t}} = \mathbf{x_{i,t}} - \mathbf{x_{i,t-1}}
```

Note: This method should only be used in combination with a position handling operation.

invert(*velocity*, *clamp=None*, ***kwargs*)
Invert the velocity if the particle is out of bounds

The velocity is inverted and shrinked. The shrinking is determined by the kwarg `z`. The default shrinking factor is `0.5`. For all velocities whose particles are out of bounds the following equation is applied: .. math:

```
\mathbf{v_{i,t}} = -z\mathbf{v_{i,t}}
```

unmodified(*velocity*, *clamp=None*, ***kwargs*)
Leaves the velocity unchanged

zero(*velocity*, *clamp=None*, ***kwargs*)
Set velocity to zero if the particle is out of bounds

1.12.3 pyswarms.topology package

This package implements various swarm topologies that may be useful as you build your own swarm implementations. Each topology can perform the following:

- Determine the best particle on a given swarm.
- Compute the next position given a current swarm position.
- Compute the velocities given a swarm configuration.

pyswarms.backend.topology.base module

Base class for Topologies

You can use this class to create your own topology. Note that every Topology should implement a way to compute the (1) best particle, the (2) next position, and the (3) next velocity given the Swarm's attributes at a given timestep. Not implementing these methods will raise an error.

In addition, this class must interface with any class found in the *pyswarms.backend.swarms.Swarm* module.

class pyswarms.backend.topology.base.**Topology**(*static*, ***kwargs*)
Bases: `abc.ABC`

__init__(*static*, ***kwargs*)
Initializes the class

compute_gbest(*swarm*)
Compute the best particle of the swarm and return the cost and position

compute_position(*swarm*)
Update the swarm's position-matrix

compute_velocity(*swarm*)
Update the swarm's velocity-matrix

pyswarms.backend.topology.star module

A Star Network Topology

This class implements a star topology. In this topology, all particles are connected to one another. This social behavior is often found in GlobalBest PSO optimizers.

class pyswarms.backend.topology.star.**Star**(*static=None*, ***kwargs*)
Bases: *pyswarms.backend.topology.base.Topology*

__init__(*static=None*, ***kwargs*)
Initializes the class

compute_gbest(*swarm*, ***kwargs*)
Update the global best using a star topology

This method takes the current pbest_pos and pbest_cost, then returns the minimum cost and position from the matrix.

```
import pyswarms.backend as P
from pyswarms.backend.swarms import Swarm
from pyswarm.backend.topology import Star

my_swarm = P.create_swarm(n_particles, dimensions)
my_topology = Star()

# Update best_cost and position
swarm.best_pos, swarm.best_cost = my_topology.compute_gbest(my_swarm)
```

Parameters **swarm** (*pyswarms.backend.swarm.Swarm*) – a Swarm instance

Returns

- *numpy.ndarray* – Best position of shape `(n_dimensions, )`
- *float* – Best cost

compute_position(*swarm*, *bounds=None*, *bh=<pyswarms.backend.handlers.BoundaryHandler object>*)
Update the position matrix

This method updates the position matrix given the current position and the velocity. If bounded, it waives updating the position.

Parameters

- **swarm** (`pyswarms.backend.swarms.Swarm`) – a Swarm instance
- **bounds** (tuple of `np.ndarray` or list (default is `None`)) – a tuple of size 2 where the first entry is the minimum bound while the second entry is the maximum bound. Each array must be of shape `(dimensions,)`.
- **bh** (`pyswarms.backend.handlers.BoundaryHandler`) – a BoundaryHandler instance

Returns New position-matrix

Return type numpy.ndarray

compute_velocity(*swarm*, *clamp=None*, *vh=<pyswarms.backend.handlers.VelocityHandler object>*, *bounds=None*)
Compute the velocity matrix

This method updates the velocity matrix using the best and current positions of the swarm. The velocity matrix is computed using the cognitive and social terms of the swarm.

A sample usage can be seen with the following:

```
import pyswarms.backend as P
from pyswarms.backend.swarm import Swarm
from pyswarms.backend.handlers import VelocityHandler
```

(continues on next page)

(continued from previous page)

```
from pyswarms.backend.topology import Star

my_swarm = P.create_swarm(n_particles, dimensions)
my_topology = Star()
my_vh = VelocityHandler(strategy="adjust")

for i in range(iters):
    # Inside the for-loop
    my_swarm.velocity = my_topology.update_velocity(my_swarm, clamp, my_vh,
    bounds)
```

Parameters

- **swarm** (pyswarms.backend.swarms.Swarm) – a Swarm instance
- **clamp** (tuple of floats (default is None)) – a tuple of size 2 where the first entry is the minimum velocity and the second entry is the maximum velocity. It sets the limits for velocity clamping.
- **vh** (pyswarms.backend.handlers.VelocityHandler) – a VelocityHandler instance
- **bounds** (tuple of np.ndarray or list (default is None)) – a tuple of size 2 where the first entry is the minimum bound while the second entry is the maximum bound. Each array must be of shape (dimensions,).

Returns Updated velocity matrix

Return type numpy.ndarray

pyswarms.backend.topology.ring module

A Ring Network Topology

This class implements a ring topology. In this topology, the particles are connected with their k nearest neighbors. This social behavior is often found in LocalBest PSO optimizers.

class pyswarms.backend.topology.ring.**Ring**(*static=False*)

Bases: *pyswarms.backend.topology.base.Topology*

__init__(*static=False*)

Initializes the class

Parameters **static** (bool (Default is False)) – a boolean that decides whether the topology is static or dynamic

compute_gbest(*swarm*, *p*, *k*, ***kwargs*)

Update the global best using a ring-like neighborhood approach

This uses the cKDTree method from scipy to obtain the nearest neighbors.

Parameters

- **swarm** (pyswarms.backend.swarms.Swarm) – a Swarm instance
- **p** (*int {1,2}*) – the Minkowski p-norm to use. 1 is the sum-of-absolute values (or L1 distance) while 2 is the Euclidean (or L2) distance.
- **k** (*int*) – number of neighbors to be considered. Must be a positive integer less than n_particles

Returns

- *numpy.ndarray* – Best position of shape (n_dimensions,)
- *float* – Best cost

compute_position(*swarm*, *bounds=None*, *bh=<pyswarms.backend.handlers.BoundaryHandler object>*)

Update the position matrix

This method updates the position matrix given the current position and the velocity. If bounded, it waives updating the position.

Parameters

- **swarm** (pyswarms.backend.swarms.Swarm) – a Swarm instance
- **bounds** (tuple of np.ndarray or list (default is None)) – a tuple of size 2 where the first entry is the minimum bound while the second entry is the maximum bound. Each array must be of shape (dimensions,).
- **bh** (pyswarms.backend.handlers.BoundaryHandler) – a BoundaryHandler instance

Returns New position-matrix

Return type numpy.ndarray

compute_velocity(*swarm*, *clamp=None*, *vh=<pyswarms.backend.handlers.VelocityHandler object>*, *bounds=None*)

Compute the velocity matrix

This method updates the velocity matrix using the best and current positions of the swarm. The velocity matrix is computed using the cognitive and social terms of the swarm.

A sample usage can be seen with the following:

```
import pyswarms.backend as P
from pyswarms.backend.swarm import Swarm
from pyswarms.backend.handlers import VelocityHandler
from pyswarms.backend.topology import Ring

my_swarm = P.create_swarm(n_particles, dimensions)
my_topology = Ring(static=False)
my_vh = VelocityHandler(strategy="invert")

for i in range(iters):
    # Inside the for-loop
    my_swarm.velocity = my_topology.update_velocity(my_swarm, clamp, my_vh,
    bounds)
```

Parameters

- **swarm** (pyswarms.backend.swarms.Swarm) – a Swarm instance
- **clamp** (tuple of floats (default is None)) – a tuple of size 2 where the first entry is the minimum velocity and the second entry is the maximum velocity. It sets the limits for velocity clamping.
- **vh** (pyswarms.backend.handlers.VelocityHandler) – a VelocityHandler instance
- **bounds** (tuple of np.ndarray or list (default is None)) – a tuple of size 2 where the first entry is the minimum bound while the second entry is the maximum bound. Each array must be of shape (dimensions,).

Returns Updated velocity matrix

Return type numpy.ndarray

pyswarms.backend.topology.von_neumann module

A Von Neumann Network Topology

This class implements a Von Neumann topology.

class pyswarms.backend.topology.von_neumann.**VonNeumann**(*static=None*)
Bases: *pyswarms.backend.topology.ring.Ring*

__init__(*static=None*)
Initializes the class

Parameters **static** (bool (Default is `False`)) – a boolean that decides whether the topology is static or dynamic

compute_gbest(*swarm*, *p*, *r*, ***kwargs*)
Updates the global best using a neighborhood approach

The Von Neumann topology inherits from the Ring topology and uses the same approach to calculate the global best. The number of neighbors is determined by the dimension and the range. This topology is always a `static` topology.

Parameters

- **swarm** (`pyswarms.backend.swarms.Swarm`) – a Swarm instance
- **r** (*int*) – range of the Von Neumann topology
- **p** (*int {1,2}*) – the Minkowski p-norm to use. 1 is the sum-of-absolute values (or L1 distance) while 2 is the Euclidean (or L2) distance.

Returns

- *numpy.ndarray* – Best position of shape `(n_dimensions, )`
- *float* – Best cost

static delannoy(*d*, *r*)
Static helper method to compute Delannoy numbers

This method computes the number of neighbours of a Von Neumann topology, i.e. a Delannoy number, dependent on the range and the dimension of the search space. The Delannoy numbers are computed recursively.

Parameters

- **d** (*int*) – dimension of the search space
- **r** (*int*) – range of the Von Neumann topology

Returns Delannoy number

Return type int

pyswarms.backend.topology.pyramid module

A Pyramid Network Topology

This class implements a pyramid topology. In this topology, the particles are connected by N-dimensional simplices.

class pyswarms.backend.topology.pyramid.**Pyramid**(*static=False*)
Bases: *pyswarms.backend.topology.base.Topology*

__init__(*static=False*)
Initialize the class

Parameters **static** (bool (Default is `False`)) – a boolean that decides whether the topology is static or dynamic

compute_gbest(*swarm*, ***kwargs*)
Update the global best using a pyramid neighborhood approach

This topology uses the `Delaunay` class from `scipy`. To prevent precision errors in the Delaunay class, custom `qhull_options` were added. Namely, `QJ0.001 Qbb Qc Qx`. The meaning of those options is explained in [qhull]. This method is used to triangulate N-dimensional space into simplices. The vertices of the simplicies consist of swarm particles. This method is adapted from the work of Lane et al.[SIS2008]

[SIS2008] J. Lane, A. Engelbrecht and J. Gain, "Particle swarm optimization with spatially meaningful neighbours," 2008 IEEE Swarm Intelligence Symposium, St. Louis, MO, 2008, pp. 1-8. doi: 10.1109/SIS.2008.4668281 [qhull] http://www.qhull.org/html/qh-optq.htm

Parameters **swarm** (`pyswarms.backend.swarms.Swarm`) – a Swarm instance

Returns

- *numpy.ndarray* – Best position of shape `(n_dimensions, )`
- *float* – Best cost

compute_position(*swarm*, *bounds=None*, *bh=<pyswarms.backend.handlers.BoundaryHandler object>*)

Update the position matrix

This method updates the position matrix given the current position and the velocity. If bounded, it waives updating the position.

Parameters

- **swarm** (`pyswarms.backend.swarms.Swarm`) – a Swarm instance
- **bounds** (tuple of `np.ndarray` or list (default is `None`)) – a tuple of size 2 where the first entry is the minimum bound while the second entry is the maximum bound. Each array must be of shape `(dimensions,)`.
- **bh** (*`a BoundaryHandler instance`*) –

Returns New position-matrix

Return type numpy.ndarray

compute_velocity(*swarm*, *clamp=None*, *vh=<pyswarms.backend.handlers.VelocityHandler object>*, *bounds=None*)

Compute the velocity matrix

This method updates the velocity matrix using the best and current positions of the swarm. The velocity matrix is computed using the cognitive and social terms of the swarm.

A sample usage can be seen with the following:

```
import pyswarms.backend as P
from pyswarms.backend.swarm import Swarm
from pyswarms.backend.handlers import VelocityHandler
from pyswarms.backend.topology import Pyramid

my_swarm = P.create_swarm(n_particles, dimensions)
my_topology = Pyramid(static=False)
my_vh = VelocityHandler(strategy="zero")

for i in range(iters):
    # Inside the for-loop
    my_swarm.velocity = my_topology.update_velocity(my_swarm, clamp, my_vh,
    bounds=bounds)
```

Parameters

- **swarm** (`pyswarms.backend.swarms.Swarm`) – a Swarm instance

- **clamp** (tuple of floats (default is None)) – a tuple of size 2 where the first entry is the minimum velocity and the second entry is the maximum velocity. It sets the limits for velocity clamping.
- **vh** (pyswarms.backend.handlers.VelocityHandler) – a VelocityHandler instance
- **bounds** (tuple of np.ndarray or list (default is None)) – a tuple of size 2 where the first entry is the minimum bound while the second entry is the maximum bound. Each array must be of shape (dimensions,).

Returns Updated velocity matrix

Return type numpy.ndarray

pyswarms.backend.topology.random module

A Random Network Topology

This class implements a random topology. All particles are connected in a random fashion.

class pyswarms.backend.topology.random.**Random**(*static=False*)
Bases: *pyswarms.backend.topology.base.Topology*

__init__(*static=False*)
Initializes the class

Parameters **static** (bool (Default is False)) – a boolean that decides whether the topology is static or dynamic

compute_gbest(*swarm*, *k*, ***kwargs*)
Update the global best using a random neighborhood approach

This uses random class from numpy to give every particle k randomly distributed, non-equal neighbors. The resulting topology is a connected graph. The algorithm to obtain the neighbors was adapted from [TSWJ2013].

[TSWJ2013] Qingjian Ni and Jianming Deng, "A New Logistic Dynamic Particle Swarm Optimization Algorithm Based on Random Topology," The Scientific World Journal, vol. 2013, Article ID 409167, 8 pages, 2013. https://doi.org/10.1155/2013/409167.

Parameters

- **swarm** (pyswarms.backend.swarms.Swarm) – a Swarm instance
- **k** (*int*) – number of neighbors to be considered. Must be a positive integer less than n_particles-1

Returns

- *numpy.ndarray* – Best position of shape (n_dimensions,)
- *float* – Best cost

compute_position(*swarm*, *bounds=None*, *bh=<pyswarms.backend.handlers.BoundaryHandler object>*)
Update the position matrix

This method updates the position matrix given the current position and the velocity. If bounded, it waives updating the position.

Parameters

- **swarm** (pyswarms.backend.swarms.Swarm) – a Swarm instance
- **bounds** (tuple of np.ndarray or list (default is None)) – a tuple of size 2 where the first entry is the minimum bound while the second entry is the maximum bound. Each array must be of shape (dimensions,).

- **bh** (pyswarms.backend.handlers.BoundaryHandler) – a BoundaryHandler instance

Returns New position-matrix

Return type numpy.ndarray

compute_velocity(*swarm*, *clamp=None*, *vh=<pyswarms.backend.handlers.VelocityHandler object>*, *bounds=None*)

Compute the velocity matrix

This method updates the velocity matrix using the best and current positions of the swarm. The velocity matrix is computed using the cognitive and social terms of the swarm.

A sample usage can be seen with the following:

```
import pyswarms.backend as P
from pyswarms.backend.swarm import Swarm
from pyswarms.backend.handlers import VelocityHandler
from pyswarms.backend.topology import Random

my_swarm = P.create_swarm(n_particles, dimensions)
my_topology = Random(static=False)
my_vh = VelocityHandler(strategy="zero")

for i in range(iters):
    # Inside the for-loop
    my_swarm.velocity = my_topology.update_velocity(my_swarm, clamp, my_vh,
    bounds)
```

Parameters

- **swarm** (pyswarms.backend.swarms.Swarm) – a Swarm instance
- **clamp** (tuple of floats (default is None)) – a tuple of size 2 where the first entry is the minimum velocity and the second entry is the maximum velocity. It sets the limits for velocity clamping.
- **vh** (pyswarms.backend.handlers.VelocityHandler) – a VelocityHandler instance
- **bounds** (tuple of np.ndarray or list (default is None)) – a tuple of size 2 where the first entry is the minimum bound while the second entry is the maximum bound. Each array must be of shape (dimensions,).

Returns Updated velocity matrix

Return type numpy.ndarray

1.12.4 pyswarms.swarms package

This package contains the Swarm class for creating your own swarm implementation. The class acts as a DataClass, holding information on the particles you have generated throughout each timestep. It offers a pre-built and flexible way of building your own swarm.

pyswarms.swarms class

class pyswarms.backend.swarms.**Swarm**(*position: numpy.ndarray, velocity: numpy.ndarray, n_particles: int = NOTHING, dimensions: int = NOTHING, options: dict = {}, pbest_pos: numpy.ndarray = NOTHING, best_pos: numpy.ndarray = array([], dtype=float64), pbest_cost: numpy.ndarray = array([], dtype=float64), best_cost: float = inf, current_cost: numpy.ndarray = array([], dtype=float64)*)

A Swarm Class

This class offers a generic swarm that can be used in most use-cases such as single-objective optimization, etc. It contains various attributes that are commonly-used in most swarm implementations.

To initialize this class, **simply supply values for the position and velocity matrix**. The other attributes are automatically filled. If you want to initialize random values, take a look at:

- *pyswarms.backend.generators.generate_swarm()*: for generating positions randomly.
- *pyswarms.backend.generators.generate_velocity()*: for generating velocities randomly.

If your swarm requires additional parameters (say c1, c2, and w in gbest PSO), simply pass them to the `options` dictionary.

As an example, say we want to create a swarm by generating particles randomly. We can use the helper methods above to do our job:

```
import pyswarms.backend as P
from pyswarms.backend.swarms import Swarm

# Let's generate a 10-particle swarm with 10 dimensions
init_positions = P.generate_swarm(n_particles=10, dimensions=10)
init_velocities = P.generate_velocity(n_particles=10, dimensions=10)
# Say, particle behavior is governed by parameters `foo` and `bar`
my_options = {'foo': 0.4, 'bar': 0.6}
# Initialize the swarm
my_swarm = Swarm(position=init_positions, velocity=init_velocities, options=my_
↪options)
```

From there, you can now use all the methods in `pyswarms.backend`. Of course, the process above has been abstracted by the method *pyswarms.backend.generators.create_swarm()* so you don't have to write the whole thing down.

position
numpy.ndarray – position-matrix at a given timestep of shape `(n_particles, dimensions)`

velocity
numpy.ndarray – velocity-matrix at a given timestep of shape `(n_particles, dimensions)`

n_particles
int (default is `position.shape[0]`) – number of particles in a swarm.

dimensions
int (default is `position.shape[1]`) – number of dimensions in a swarm.

options
dict (default is empty dictionary) – various options that govern a swarm's behavior.

pbest_pos
numpy.ndarray (default is `None`) – personal best positions of each particle of shape `(n_particles, dimensions)`

best_pos
numpy.ndarray (default is empty array) – best position found by the swarm of shape `(dimensions,`

) for the Star`topology and :code:`(dimensions, particles) for the other topologies

pbest_cost
numpy.ndarray (default is empty array) – personal best costs of each particle of shape (n_particles,)

best_cost
float (default is np.inf) – best cost found by the swarm

current_cost
numpy.ndarray (default is empty array) – the current cost found by the swarm of shape (n_particles, dimensions)

1.13 Base Classes

The base classes are inherited by various PSO implementations throughout the library. It supports a simple skeleton to construct a customized PSO algorithm.

1.13.1 pyswarms.base package

The *pyswarms.base* module implements base swarm classes to implement variants of particle swarm optimization.

pyswarms.base module

Base class for single-objective Particle Swarm Optimization implementations.

All methods here are abstract and raise a NotImplementedError when not used. When defining your own swarm implementation, create another class,

```
>>> class MySwarm(SwarmBase):
>>>     def __init__(self):
>>>         super(MySwarm, self).__init__()
```

and define all the necessary methods needed.

As a guide, check the global best and local best implementations in this package.

Note: Regarding options, it is highly recommended to include parameters used in position and velocity updates as keyword arguments. For parameters that affect the topology of the swarm, it may be much better to have them as positional arguments.

See also:

pyswarms.single.global_best global-best PSO implementation

pyswarms.single.local_best local-best PSO implementation

pyswarms.single.general_optimizer a more general PSO implementation with a custom topology

class pyswarms.base.base_single.**SwarmOptimizer**(*n_particles*, *dimensions*, *options*, *bounds=None*, *velocity_clamp=None*, *center=1.0*, *ftol=-inf*, *init_pos=None*)

Bases: abc.ABC

__init__(*n_particles*, *dimensions*, *options*, *bounds=None*, *velocity_clamp=None*, *center=1.0*, *ftol=-inf*, *init_pos=None*)
Initialize the swarm

Creates a Swarm class depending on the values initialized

n_particles
int – number of particles in the swarm.

dimensions
int – number of dimensions in the space.

options
dict with keys `{'c1', 'c2', 'w'}` – a dictionary containing the parameters for the specific optimization technique
- **c1** [float] cognitive parameter
- **c2** [float] social parameter
- **w** [float] inertia parameter

bounds
tuple of `np.ndarray` (default is `None`) – a tuple of size 2 where the first entry is the minimum bound while the second entry is the maximum bound. Each array must be of shape `(dimensions,)`.

velocity_clamp
tuple (default is `None`) – a tuple of size 2 where the first entry is the minimum velocity and the second entry is the maximum velocity. It sets the limits for velocity clamping.

center
list (default is `None`) – an array of size `dimensions`

ftol
float – relative error in objective_func(best_pos) acceptable for convergence

_abc_impl = <_abc_data object>

_populate_history(*hist*)
Populate all history lists

The `cost_history`, `mean_pbest_history`, and `neighborhood_best` is expected to have a shape of `(iters,)`,on the other hand, the `pos_history` and `velocity_history` are expected to have a shape of `(iters, n_particles, dimensions)`

Parameters **hist** (*namedtuple*) – Must be of the same type as self.ToHistory

optimize(*objective_func*, *iters*, *n_processes=None*, ***kwargs*)
Optimize the swarm for a number of iterations

Performs the optimization to evaluate the objective function `objective_func` for a number of iterations `iter`.

Parameters

- **objective_func** (*function*) – objective function to be evaluated
- **iters** (*int*) – number of iterations
- **n_processes** (*int*) – number of processes to use for parallel particle evaluation (default: None = no parallelization)
- **kwargs** (*dict*) – arguments for objective function

Raises `NotImplementedError` – When this method is not implemented.

reset()
Reset the attributes of the optimizer

All variables/atributes that will be re-initialized when this method is defined here. Note that this method can be called twice: (1) during initialization, and (2) when this is called from an instance.

It is good practice to keep the number of resettable attributes at a minimum. This is to prevent spamming the same object instance with various swarm definitions.

Normally, swarm definitions are as atomic as possible, where each type of swarm is contained in its own instance. Thus, the following attributes are the only ones recommended to be resettable:

- Swarm position matrix (self.pos)
- Velocity matrix (self.pos)
- Best scores and positions (gbest_cost, gbest_pos, etc.)

Otherwise, consider using positional arguments.

Base class for single-objective discrete Particle Swarm Optimization implementations.

All methods here are abstract and raises a `NotImplementedError` when not used. When defining your own swarm implementation, create another class,

```
>>> class MySwarm(DiscreteSwarmOptimizer):
>>>     def __init__(self):
>>>         super(MySwarm, self).__init__()
```

and define all the necessary methods needed.

As a guide, check the discrete PSO implementations in this package.

Note: Regarding `options`, it is highly recommended to include parameters used in position and velocity updates as keyword arguments. For parameters that affect the topology of the swarm, it may be much better to have them as positional arguments.

See also:

pyswarms.discrete.binary binary PSO implementation

class pyswarms.base.base_discrete.**DiscreteSwarmOptimizer**(*n_particles*, *dimensions*, *binary*, *options*, *velocity_clamp=None*, *init_pos=None*, *ftol=-inf*)

Bases: `abc.ABC`

__init__(*n_particles*, *dimensions*, *binary*, *options*, *velocity_clamp=None*, *init_pos=None*, *ftol=-inf*)

Initialize the swarm.

Creates a `numpy.ndarray` of positions depending on the number of particles needed and the number of dimensions. The initial positions of the particles depends on the argument `binary`, which governs if a binary matrix will be produced.

n_particles
int – number of particles in the swarm.

dimensions
int – number of dimensions in the space.

binary
boolean – a trigger to generate a binary matrix for the swarm's initial positions. When passed with a `False` value, random integers from 0 to `dimensions` are generated.

options
dict with keys `{'c1', 'c2', 'w'}` – a dictionary containing the parameters for the specific optimization technique
- **c1** [float] cognitive parameter
- **c2** [float] social parameter

- **w** [float] inertia parameter

velocity_clamp
tuple (default is `None`) – a tuple of size 2 where the first entry is the minimum velocity and the second entry is the maximum velocity. It sets the limits for velocity clamping.

options
dict – a dictionary containing the parameters for a specific optimization technique

_abc_impl = <_abc_data object>

_populate_history(*hist*)
Populate all history lists

The `cost_history`, `mean_pbest_history`, and `neighborhood_best` is expected to have a shape of `(iters,)`,on the other hand, the `pos_history` and `velocity_history` are expected to have a shape of `(iters, n_particles, dimensions)`

Parameters **hist** (*namedtuple*) – Must be of the same type as self.ToHistory

optimize(*objective_func*, *iters*, *n_processes=None*, ***kwargs*)
Optimize the swarm for a number of iterations

Performs the optimization to evaluate the objective function `objective_func` for a number of iterations `iter.`

Parameters

- **objective_func** (*function*) – objective function to be evaluated
- **iters** (*int*) – number of iterations
- **n_processes** (*int*) – number of processes to use for parallel particle evaluation (default: None = no parallelization)
- **kwargs** (*dict*) – arguments for objective function

Raises `NotImplementedError` – When this method is not implemented.

reset()
Reset the attributes of the optimizer

All variables/atributes that will be re-initialized when this method is defined here. Note that this method can be called twice: (1) during initialization, and (2) when this is called from an instance.

It is good practice to keep the number of resettable attributes at a minimum. This is to prevent spamming the same object instance with various swarm definitions.

Normally, swarm definitions are as atomic as possible, where each type of swarm is contained in its own instance. Thus, the following attributes are the only ones recommended to be resettable:

- Swarm position matrix (self.pos)
- Velocity matrix (self.pos)
- Best scores and positions (gbest_cost, gbest_pos, etc.)

Otherwise, consider using positional arguments.

1.14 Optimizers

Off-the-shelf implementations of standard algorithms. Includes classics such as global-best and local-best. Useful for quick-and-easy optimization problems.

1.14.1 pyswarms.single package

The *pyswarms.single* module implements various techniques in continuous single-objective optimization. These require only one objective function that can be optimized in a continuous space.

Note: PSO algorithms scale with the search space. This means that, by using larger boundaries, the final results are getting larger as well.

Note: Please keep in mind that Python has a biggest float number. So using large boundaries in combination with exponentiation or multiplication can lead to an `OverflowError`.

---

pyswarms.single.global_best module

A Global-best Particle Swarm Optimization (gbest PSO) algorithm.

It takes a set of candidate solutions, and tries to find the best solution using a position-velocity update method. Uses a star-topology where each particle is attracted to the best performing particle.

The position update can be defined as:

$$x_i(t+1) = x_i(t) + v_i(t+1)$$

Where the position at the current timestep t is updated using the computed velocity at $t+1$. Furthermore, the velocity update is defined as:

$$v_{ij}(t+1) = m * v_{ij}(t) + c_1 r_{1j}(t)[y_{ij}(t) x_{ij}(t)] + c_2 r_{2j}(t)[\hat{y}_j(t) x_{ij}(t)]$$

Here, $c1$ and $c2$ are the cognitive and social parameters respectively. They control the particle's behavior given two choices: (1) to follow its *personal best* or (2) follow the swarm's *global best* position. Overall, this dictates if the swarm is explorative or exploitative in nature. In addition, a parameter w controls the inertia of the swarm's movement.

An example usage is as follows:

```
import pyswarms as ps
from pyswarms.utils.functions import single_obj as fx

# Set-up hyperparameters
options = {'c1': 0.5, 'c2': 0.3, 'w':0.9}

# Call instance of GlobalBestPSO
optimizer = ps.single.GlobalBestPSO(n_particles=10, dimensions=2,
                                    options=options)

# Perform optimization
stats = optimizer.optimize(fx.sphere, iters=100)
```

This algorithm was adapted from the earlier works of J. Kennedy and R.C. Eberhart in Particle Swarm Optimization *[IJCNN1995]*.

class pyswarms.single.global_best.**GlobalBestPSO**(*n_particles*, *dimensions*, *options*, *bounds=None*, *bh_strategy='periodic'*, *velocity_clamp=None*, *vh_strategy='unmodified'*, *center=1.0*, *ftol=-inf*, *init_pos=None*)

Bases: *pyswarms.base.base_single.SwarmOptimizer*

__init__(*n_particles*, *dimensions*, *options*, *bounds=None*, *bh_strategy='periodic'*, *velocity_clamp=None*, *vh_strategy='unmodified'*, *center=1.0*, *ftol=-inf*, *init_pos=None*)
Initialize the swarm

n_particles
int – number of particles in the swarm.

dimensions
int – number of dimensions in the space.

options
dict with keys `{'c1', 'c2', 'w'}` – a dictionary containing the parameters for the specific optimization technique.
- **c1** [float] cognitive parameter
- **c2** [float] social parameter
- **w** [float] inertia parameter

bounds
tuple of `np.ndarray` (default is `None`) – a tuple of size 2 where the first entry is the minimum bound while the second entry is the maximum bound. Each array must be of shape `(dimensions,)`.

bh_strategy
String – a strategy for the handling of out-of-bounds particles.

velocity_clamp
tuple (default is `None`) – a tuple of size 2 where the first entry is the minimum velocity and the second entry is the maximum velocity. It sets the limits for velocity clamping.

vh_strategy
String – a strategy for the handling of the velocity of out-of-bounds particles.

center
list (default is `None`) – an array of size `dimensions`

ftol
float – relative error in objective_func(best_pos) acceptable for convergence

init_pos
`numpy.ndarray` (default is `None`) – option to explicitly set the particles' initial positions. Set to `None` if you wish to generate the particles randomly.

optimize(*objective_func*, *iters*, *n_processes=None*, ***kwargs*)
Optimize the swarm for a number of iterations

Performs the optimization to evaluate the objective function `f` for a number of iterations `iter.`

Parameters
- **objective_func** (*function*) – objective function to be evaluated
- **iters** (*int*) – number of iterations
- **n_processes** (*int*) – number of processes to use for parallel particle evaluation (default: None = no parallelization)
- **kwargs** (*dict*) – arguments for the objective function

Returns the global best cost and the global best position.

Return type tuple

pyswarms.single.local_best module

A Local-best Particle Swarm Optimization (lbest PSO) algorithm.

Similar to global-best PSO, it takes a set of candidate solutions, and finds the best solution using a position-velocity update method. However, it uses a ring topology, thus making the particles attracted to its corresponding neighborhood.

The position update can be defined as:

$$x_i(t+1) = x_i(t) + v_i(t+1)$$

Where the position at the current timestep t is updated using the computed velocity at $t+1$. Furthermore, the velocity update is defined as:

$$v_{ij}(t+1) = m * v_{ij}(t) + c_1 r_{1j}(t)[y_{ij}(t) x_{ij}(t)] + c_2 r_{2j}(t)[\hat{y}_j(t) x_{ij}(t)]$$

However, in local-best PSO, a particle doesn't compare itself to the overall performance of the swarm. Instead, it looks at the performance of its nearest-neighbours, and compares itself with them. In general, this kind of topology takes much more time to converge, but has a more powerful explorative feature.

In this implementation, a neighbor is selected via a k-D tree imported from `scipy`. Distance are computed with either the L1 or L2 distance. The nearest-neighbours are then queried from this k-D tree. They are computed for every iteration.

An example usage is as follows:

```
import pyswarms as ps
from pyswarms.utils.functions import single_obj as fx

# Set-up hyperparameters
options = {'c1': 0.5, 'c2': 0.3, 'w': 0.9, 'k': 3, 'p': 2}

# Call instance of LBestPSO with a neighbour-size of 3 determined by
# the L2 (p=2) distance.
optimizer = ps.single.LocalBestPSO(n_particles=10, dimensions=2,
                                   options=options)

# Perform optimization
stats = optimizer.optimize(fx.sphere, iters=100)
```

This algorithm was adapted from one of the earlier works of J. Kennedy and R.C. Eberhart in Particle Swarm Optimization *[IJCNN1995] [MHS1995]*

class pyswarms.single.local_best.**LocalBestPSO**(*n_particles*, *dimensions*, *options*, *bounds=None*, *bh_strategy='periodic'*, *velocity_clamp=None*, *vh_strategy='unmodified'*, *center=1.0*, *ftol=-inf*, *init_pos=None*, *static=False*)

Bases: *pyswarms.base.base_single.SwarmOptimizer*

__init__(*n_particles*, *dimensions*, *options*, *bounds=None*, *bh_strategy='periodic'*, *velocity_clamp=None*, *vh_strategy='unmodified'*, *center=1.0*, *ftol=-inf*, *init_pos=None*, *static=False*)

Initialize the swarm

n_particles
int – number of particles in the swarm.

dimensions
int – number of dimensions in the space.

bounds
tuple of np.ndarray, optional (default is `None`) – a tuple of size 2 where the first entry is the minimum bound while the second entry is the maximum bound. Each array must be of shape `(dimensions,)`.

bh_strategy
String – a strategy for the handling of out-of-bounds particles.

velocity_clamp
tuple (default is `(0,1)`) – a tuple of size 2 where the first entry is the minimum velocity and the second entry is the maximum velocity. It sets the limits for velocity clamping.

vh_strategy
String – a strategy for the handling of the velocity of out-of-bounds particles.

center
list (default is `None`) – an array of size `dimensions`

ftol
float – relative error in objective_func(best_pos) acceptable for convergence

options
dict with keys `{'c1', 'c2', 'w', 'k', 'p'}` – a dictionary containing the parameters for the specific optimization technique
- **c1** [float] cognitive parameter
- **c2** [float] social parameter
- **w** [float] inertia parameter
- **k** [int] number of neighbors to be considered. Must be a positive integer less than `n_particles`
- **p: int {1,2}** the Minkowski p-norm to use. 1 is the sum-of-absolute values (or L1 distance) while 2 is the Euclidean (or L2) distance.

init_pos
`numpy.ndarray` (default is `None`) – option to explicitly set the particles' initial positions. Set to `None` if you wish to generate the particles randomly.

static
bool (Default is `False`) – a boolean that decides whether the Ring topology used is static or dynamic

_abc_impl = <_abc_data object>

optimize(*objective_func*, *iters*, *n_processes=None*, ***kwargs*)
Optimize the swarm for a number of iterations

Performs the optimization to evaluate the objective function `f` for a number of iterations `iter.`

Parameters

- **objective_func** (*function*) – objective function to be evaluated
- **iters** (*int*) – number of iterations
- **n_processes** (*int*) – number of processes to use for parallel particle evaluation (default: None = no parallelization)
- **kwargs** (*dict*) – arguments for the objective function

Returns the local best cost and the local best position among the swarm.

Return type tuple

pyswarms.single.general_optimizer module

A general Particle Swarm Optimization (general PSO) algorithm.

It takes a set of candidate solutions, and tries to find the best solution using a position-velocity update method. Uses a user specified topology.

The position update can be defined as:

$$x_i(t+1) = x_i(t) + v_i(t+1)$$

Where the position at the current timestep t is updated using the computed velocity at $t + 1$. Furthermore, the velocity update is defined as:

$$v_{ij}(t+1) = m * v_{ij}(t) + c_1 r_{1j}(t)[y_{ij}(t)x_{ij}(t)] + c_2 r_{2j}(t)[\hat{y}_j(t)x_{ij}(t)]$$

Here, $c1$ and $c2$ are the cognitive and social parameters respectively. They control the particle's behavior given two choices: (1) to follow its *personal best* or (2) follow the swarm's *global best* position. Overall, this dictates if the swarm is explorative or exploitative in nature. In addition, a parameter w controls the inertia of the swarm's movement.

An example usage is as follows:

```
import pyswarms as ps
from pyswarms.backend.topology import Pyramid
from pyswarms.utils.functions import single_obj as fx

# Set-up hyperparameters and topology
options = {'c1': 0.5, 'c2': 0.3, 'w':0.9}
my_topology = Pyramid(static=False)

# Call instance of GlobalBestPSO
optimizer = ps.single.GeneralOptimizerPSO(n_particles=10, dimensions=2,
                                    options=options, topology=my_topology)

# Perform optimization
stats = optimizer.optimize(fx.sphere, iters=100)
```

This algorithm was adapted from the earlier works of J. Kennedy and R.C. Eberhart in Particle Swarm Optimization *[IJCNN1995]*.

class pyswarms.single.general_optimizer.**GeneralOptimizerPSO**(*n_particles*, *dimensions*, *options*, *topology*, *bounds=None*, *bh_strategy='periodic'*, *velocity_clamp=None*, *vh_strategy='unmodified'*, *center=1.0*, *ftol=-inf*, *init_pos=None*)

Bases: *pyswarms.base.base_single.SwarmOptimizer*

__init__(*n_particles*, *dimensions*, *options*, *topology*, *bounds=None*, *bh_strategy='periodic'*, *velocity_clamp=None*, *vh_strategy='unmodified'*, *center=1.0*, *ftol=-inf*, *init_pos=None*)

Initialize the swarm

n_particles
int – number of particles in the swarm.

dimensions
int – number of dimensions in the space.

options
dict with keys `{'c1', 'c2', 'w'}` or `{'c1', -- 'c2', 'w', 'k', 'p'}` a dictionary containing the parameters for the specific optimization technique.
- **c1** [float] cognitive parameter
- **c2** [float] social parameter
- **w** [float] inertia parameter

if used with the `Ring`, `VonNeumann` or `Random` topology the additional parameter k must be included * k : int
number of neighbors to be considered. Must be a positive integer less than `n_particles`

if used with the `Ring` topology the additional parameters k and p must be included * p: int {1,2}
the Minkowski p-norm to use. 1 is the sum-of-absolute values (or L1 distance) while 2 is the Euclidean (or L2) distance.
if used with the `VonNeumann` topology the additional parameters p and r must be included * r: int
the range of the VonNeumann topology. This is used to determine the number of neighbours in the topology.

topology
pyswarms.backend.topology.Topology – a `Topology` object that defines the topology to use in the optimization process. The currently available topologies are:
- **Star** All particles are connected
- **Ring (static and dynamic)** Particles are connected to the k nearest neighbours
- **VonNeumann** Particles are connected in a VonNeumann topology
- **Pyramid (static and dynamic)** Particles are connected in N-dimensional simplices
- **Random (static and dynamic)** Particles are connected to k random particles

Static variants of the topologies remain with the same neighbours over the course of the optimization. Dynamic variants calculate new neighbours every time step.

bounds
tuple of `np.ndarray` (default is `None`) – a tuple of size 2 where the first entry is the minimum bound while the second entry is the maximum bound. Each array must be of shape `(dimensions,)`.

bh_strategy
String – a strategy for the handling of out-of-bounds particles.

velocity_clamp
tuple (default is `None`) – a tuple of size 2 where the first entry is the minimum velocity and the second entry is the maximum velocity. It sets the limits for velocity clamping.

vh_strategy
String – a strategy for the handling of the velocity of out-of-bounds particles.

center
list (default is `None`) – an array of size `dimensions`

ftol
float – relative error in objective_func(best_pos) acceptable for convergence

init_pos
`numpy.ndarray` (default is `None`) – option to explicitly set the particles' initial positions. Set to `None` if you wish to generate the particles randomly.

_abc_impl = <_abc_data object>

optimize(*objective_func*, *iters*, *n_processes=None*, ***kwargs*)
Optimize the swarm for a number of iterations

Performs the optimization to evaluate the objective function `f` for a number of iterations `iter.`

Parameters
- **objective_func** (*function*) – objective function to be evaluated
- **iters** (*int*) – number of iterations
- **n_processes** (*int*) – number of processes to use for parallel particle evaluation (default: None = no parallelization)
- **kwargs** (*dict*) – arguments for the objective function

Returns the global best cost and the global best position.

Return type tuple

1.14.2 pyswarms.discrete package

The *pyswarms.discrete* module implements various techniques in discrete optimization. These are techniques that can be applied to a discrete search-space.

pyswarms.discrete.binary module

A Binary Particle Swarm Optimization (binary PSO) algorithm.

It takes a set of candidate solutions, and tries to find the best solution using a position-velocity update method. Unlike `pyswarms.single.gb` and `pyswarms.single.lb`, this technique is often applied to discrete binary problems such as job-shop scheduling, sequencing, and the like.

The update rule for the velocity is still similar, as shown in the proceeding equation:

$$v_{ij}(t+1) = m * v_{ij}(t) + c_1 r_{1j}(t)[y_{ij}(t) x_{ij}(t)] + c_2 r_{2j}(t)[\hat{y}_j(t) x_{ij}(t)]$$

For the velocity update rule, a particle compares its current position with respect to its neighbours. The nearest neighbours are being determined by a kD-tree given a distance metric, similar to local-best PSO. The neighbours are computed for every iteration. However, this whole behavior can be modified into a global-best PSO by changing the nearest neighbours equal to the number of particles in the swarm. In this case, all particles see each other, and thus a global best particle can be established.

In addition, one notable change for binary PSO is that the position update rule is now decided upon by the following case expression:

$$X_{ij}(t+1) = \left\{ \begin{array}{ll} 0, & \text{if rand()} \geq S(v_{ij}(t+1)) \\ 1, & \text{if rand()} < S(v_{ij}(t+1)) \end{array} \right\}$$

Where the function $S(x)$ is the sigmoid function defined as:

$$S(x) = \frac{1}{1 + e^{-x}}$$

This enables the algorithm to output binary positions rather than a stream of continuous values as seen in global-best or local-best PSO.

This algorithm was adapted from the standard Binary PSO work of J. Kennedy and R.C. Eberhart in Particle Swarm Optimization *[SMC1997]*.

class pyswarms.discrete.binary.**BinaryPSO**(*n_particles*, *dimensions*, *options*, *init_pos=None*, *velocity_clamp=None*, *vh_strategy='unmodified'*, *ftol=-inf*)

Bases: *pyswarms.base.base_discrete.DiscreteSwarmOptimizer*

__init__(*n_particles*, *dimensions*, *options*, *init_pos=None*, *velocity_clamp=None*, *vh_strategy='unmodified'*, *ftol=-inf*)

Initialize the swarm

n_particles
int – number of particles in the swarm.

dimensions
int – number of dimensions in the space.

options
dict with keys `{'c1', 'c2', 'k', 'p'}` – a dictionary containing the parameters for the specific optimization technique

- **c1** [float] cognitive parameter
- **c2** [float] social parameter
- **w** [float] inertia parameter
- **k** [int] number of neighbors to be considered. Must be a positive integer less than `n_particles`

- **p: int {1,2}** the Minkowski p-norm to use. 1 is the sum-of-absolute values (or L1 distance) while 2 is the Euclidean (or L2) distance.

init_pos
numpy.ndarray (default is None) – option to explicitly set the particles' initial positions. Set to None if you wish to generate the particles randomly.

velocity_clamp
tuple (default is None) – a tuple of size 2 where the first entry is the minimum velocity and the second entry is the maximum velocity. It sets the limits for velocity clamping.

vh_strategy
String – a strategy for the handling of the velocity of out-of-bounds particles. Only the "unmodified" and the "adjust" strategies are allowed.

ftol
float – relative error in objective_func(best_pos) acceptable for convergence

optimize(*objective_func*, *iters*, *n_processes=None*, ***kwargs*)
Optimize the swarm for a number of iterations

Performs the optimization to evaluate the objective function f for a number of iterations iter.

Parameters

- **objective_func** (*function*) – objective function to be evaluated
- **iters** (*int*) – number of iterations
- **n_processes** (*int*) – number of processes to use for parallel particle evaluation (default: None = no parallelization)
- **kwargs** (*dict*) – arguments for objective function

Returns the local best cost and the local best position among the swarm.

Return type tuple

1.15 Utilities

This includes various utilities to help in optimization. Some utilities include benchmark objective functions, hyperparameter search, and plotting functionalities.

1.15.1 pyswarms.utils.decorators package

The pyswarms.decorators module implements a decorator that can be used to simplify the task of writing the cost function for an optimization run. The decorator can be directly called by using @pyswarms.cost.

pyswarms.utils.decorators.**cost**(*cost_func*)
A decorator for the cost function

This decorator allows the creation of much simpler cost functions. Instead of writing a cost function that returns a shape of (n_particles, 0) it enables the usage of shorter and simpler cost functions that directly return the cost. A simple example might be:

The decorator expects your cost function to use a d-dimensional array (where d is the number of dimensions for the optimization) as and argument.

Note: Some numpy functions return a np.ndarray with single values in it. Be aware of the fact that without unpacking the value the optimizer will raise an exception.

Parameters **cost_func** (*callable*) – A callable object that can be used as cost function in the optimization (must return a `float` or an `int`).

Returns The vectorized output for all particles as defined by `cost_func`

Return type callable

1.15.2 pyswarms.utils.functions package

The mod:*pyswarms.utils.functions* module implements various test functions for optimization.

pyswarms.utils.functions.single_obj module

single_obj.py: collection of single-objective functions

All objective functions `obj_func()` must accept a `(numpy.ndarray)` with shape `(n_particles, dimensions)`. Thus, each row represents a particle, and each column represents its position on a specific dimension of the search-space.

In this context, `obj_func()` must return an array `j` of size `(n_particles, )` that contains all the computed fitness for each particle.

Whenever you make changes to this file via an implementation of a new objective function, be sure to perform unittesting in order to check if all functions implemented adheres to the design pattern stated above.

Function list: - Ackley's, ackley - Beale, beale - Booth, booth - Bukin's No 6, bukin6 - Cross-in-Tray, crossintray - Easom, easom - Eggholder, eggholder - Goldstein, goldstein - Himmelblau's, himmelblau - Holder Table, holdertable - Levi, levi - Matyas, matyas - Rastrigin, rastrigin - Rosenbrock, rosenbrock - Schaffer No 2, schaffer2 - Sphere, sphere - Three Hump Camel, threehump

`pyswarms.utils.functions.single_obj.`**`ackley`**`(x)`

Ackley's objective function.

Has a global minimum of *0* at `f(0,0,...,0)` with a search domain of [-32, 32]

Parameters **x** (*numpy.ndarray*) – set of inputs of shape `(n_particles, dimensions)`

Returns computed cost of size `(n_particles, )`

Return type numpy.ndarray

ValueError When the input is out of bounds with respect to the function domain

`pyswarms.utils.functions.single_obj.`**`beale`**`(x)`

Beale objective function.

Only takes two dimensions and has a global minimum of *0* at `f([3,0.5])` Its domain is bounded between `[-4.5, 4.5]`

Parameters **x** (*numpy.ndarray*) – set of inputs of shape `(n_particles, dimensions)`

Returns computed cost of size `(n_particles, )`

Return type numpy.ndarray

Raises

- `IndexError` – When the input dimensions is greater than what the function allows
- `ValueError` – When the input is out of bounds with respect to the function domain

pyswarms.utils.functions.single_obj.**booth**(*x*)
Booth's objective function.

Only takes two dimensions and has a global minimum of *0* at `f([1,3])`. Its domain is bounded between `[-10, 10]`

Parameters **x** (*numpy.ndarray*) – set of inputs of shape `(n_particles, dimensions)`

Returns computed cost of size `(n_particles, )`

Return type numpy.ndarray

Raises

- `IndexError` – When the input dimensions is greater than what the function allows
- `ValueError` – When the input is out of bounds with respect to the function domain

pyswarms.utils.functions.single_obj.**bukin6**(*x*)
Bukin N. 6 Objective Function

Only takes two dimensions and has a global minimum of *0* at `f([-10,1])`. Its coordinates are bounded by:

- x[:,0] must be within [-15, -5]
- x[:,1] must be within [-3, 3]

Parameters **x** (*numpy.ndarray*) – set of inputs of shape `(n_particles, dimensions)`

Returns computed cost of size `(n_particles, )`

Return type numpy.ndarray

Raises

- `IndexError` – When the input dimensions is greater than what the function allows
- `ValueError` – When the input is out of bounds with respect to the function domain

pyswarms.utils.functions.single_obj.**crossintray**(*x*)
Cross-in-tray objective function.

Only takes two dimensions and has a four equal global minimums of *-2.06261* at `f([1.34941, -1.34941])`, `f([1.34941, 1.34941])`, `f([-1.34941, 1.34941])`, and `f([-1.34941, -1.34941])`.

Its coordinates are bounded within `[-10,10]`.

Best visualized in the full domain and a range of `[-2.0, -0.5]`.

Parameters **x** (*numpy.ndarray*) – set of inputs of shape `(n_particles, dimensions)`

Returns computed cost of size `(n_particles, )`

Return type numpy.ndarray

Raises

- `IndexError` – When the input dimensions is greater than what the function allows
- `ValueError` – When the input is out of bounds with respect to the function domain

pyswarms.utils.functions.single_obj.**easom**(*x*)
Easom objective function.

Only takes two dimensions and has a global minimum of *-1* at `f([pi, pi])`. Its coordinates are bounded within `[-100,100]`.

Best visualized in the domain of `[-5, 5]` and a range of `[-1, 0.2]`.

Parameters **x** (*numpy.ndarray*) – set of inputs of shape `(n_particles, dimensions)`

Returns computed cost of size `(n_particles, )`

Return type numpy.ndarray

Raises

- `IndexError` – When the input dimensions is greater than what the function allows
- `ValueError` – When the input is out of bounds with respect to the function domain

pyswarms.utils.functions.single_obj.**eggholder**(*x*)

Eggholder objective function.

Only takes two dimensions and has a global minimum of *-959.6407* at `f([512, 404.3219])`. Its coordinates are bounded within `[-512, 512]`.

Best visualized in the full domain and a range of `[-1000, 1000]`.

Parameters **x** (*numpy.ndarray*) – set of inputs of shape `(n_particles, dimensions)`

Returns computed cost of size `(n_particles, )`

Return type numpy.ndarray

Raises

- `IndexError` – When the input dimensions is greater than what the function allows
- `ValueError` – When the input is out of bounds with respect to the function domain

pyswarms.utils.functions.single_obj.**goldstein**(*x*)

Goldstein-Price's objective function.

Only takes two dimensions and has a global minimum at `f([0,-1])`. Its domain is bounded between `[-2, 2]`

Best visualized in the domain of `[-1.3,1.3]` and range `[-1,8000]`

Parameters **x** (*numpy.ndarray*) – set of inputs of shape `(n_particles, dimensions)`

Returns computed cost of size `(n_particles, )`

Return type numpy.ndarray

Raises

- `IndexError` – When the input dimensions is greater than what the function allows
- `ValueError` – When the input is out of bounds with respect to the function domain

pyswarms.utils.functions.single_obj.**himmelblau**(*x*)

Himmelblau's objective function

Only takes two dimensions and has a four equal global minimums of zero at `f([3.0,2.0])`, `f([-2.805118,3.131312])`, `f([-3.779310,-3.283186])`, and `f([3.584428,-1.848126])`.

Its coordinates are bounded within `[-5,5]`.

Best visualized with the full domain and a range of `[0,1000]`

Parameters **x** (*numpy.ndarray*) – set of inputs of shape `(n_particles, dimensions)`

Returns computed cost of size `(n_particles, )`

Return type numpy.ndarray

Raises

- `IndexError` – When the input dimensions is greater than what the function allows
- `ValueError` – When the input is out of bounds with respect to the function domain

`pyswarms.utils.functions.single_obj.`**`holdertable`**`(x)`

Holder Table objective function

Only takes two dimensions and has a four equal global minimums of *-19.2085* at `f([8.05502, 9.66459])`, `f([-8.05502, 9.66459])`, `f([8.05502, -9.66459])`, and `f([-8.05502, -9.66459])`.

Its coordinates are bounded within `[-10, 10]`.

Best visualized with the full domain and a range of `[-20, 0]`

Parameters **x** (*numpy.ndarray*) – set of inputs of shape `(n_particles, dimensions)`

Returns computed cost of size `(n_particles, )`

Return type numpy.ndarray

Raises

- `IndexError` – When the input dimensions is greater than what the function allows
- `ValueError` – When the input is out of bounds with respect to the function domain

`pyswarms.utils.functions.single_obj.`**`levi`**`(x)`

Levi objective function

Only takes two dimensions and has a global minimum at `f([1,1])`. Its coordinates are bounded within `[-10,10]`.

Parameters **x** (*numpy.ndarray*) – set of inputs of shape `(n_particles, dimensions)`

Returns computed cost of size `(n_particles, )`

Return type numpy.ndarray

Raises

- `IndexError` – When the input dimensions is greater than what the function allows
- `ValueError` – When the input is out of bounds with respect to the function domain

`pyswarms.utils.functions.single_obj.`**`matyas`**`(x)`

Matyas objective function

Only takes two dimensions and has a global minimum at `f([0,0])`. Its coordinates are bounded within `[-10,10]`.

Parameters **x** (*numpy.ndarray*) – set of inputs of shape `(n_particles, dimensions)`

Returns

Return type numpy.ndarray

`pyswarms.utils.functions.single_obj.`**`rastrigin`**`(x)`

Rastrigin objective function.

Has a global minimum at `f(0,0,...,0)` with a search domain of `[-5.12, 5.12]`

Parameters **x** (*numpy.ndarray*) – set of inputs of shape `(n_particles, dimensions)`

Returns computed cost of size `(n_particles, )`

Return type numpy.ndarray

Raises `ValueError` – When the input is out of bounds with respect to the function domain

pyswarms.utils.functions.single_obj.**rosenbrock**(*x*)
Rosenbrock objective function.

Also known as the Rosenbrock's valley or Rosenbrock's banana function. Has a global minimum of `np.ones(dimensions)` where `dimensions` is `x.shape[1]`. The search domain is `[-inf, inf]`.

Parameters **x** (*numpy.ndarray*) – set of inputs of shape `(n_particles, dimensions)`

Returns computed cost of size `(n_particles, )`

Return type numpy.ndarray

pyswarms.utils.functions.single_obj.**schaffer2**(*x*)
Schaffer N.2 objective function

Only takes two dimensions and has a global minimum at `f([0,0])`. Its coordinates are bounded within `[-100,100]`.

Parameters **x** (*numpy.ndarray*) – set of inputs of shape `(n_particles, dimensions)`

Returns computed cost of size `(n_particles, )`

Return type numpy.ndarray

Raises

- `IndexError` – When the input dimensions is greater than what the function allows
- `ValueError` – When the input is out of bounds with respect to the function domain

pyswarms.utils.functions.single_obj.**sphere**(*x*)
Sphere objective function.

Has a global minimum at 0 and with a search domain of `[-inf, inf]`

Parameters **x** (*numpy.ndarray*) – set of inputs of shape `(n_particles, dimensions)`

Returns computed cost of size `(n_particles, )`

Return type numpy.ndarray

pyswarms.utils.functions.single_obj.**threehump**(*x*)
Three-hump camel objective function

Only takes two dimensions and has a global minimum of *0* at `f([0, 0])`. Its coordinates are bounded within `[-5, 5]`.

Best visualized in the full domin and a range of `[0, 2000]`.

Parameters **x** (*numpy.ndarray*) – set of inputs of shape `(n_particles, dimensions)`

Returns computed cost of size `(n_particles, )`

Return type numpy.ndarray

Raises

- `IndexError` – When the input dimensions is greater than what the function allows
- `ValueError` – When the input is out of bounds with respect to the function domain

1.15.3 pyswarms.utils.plotters package

The mod:*pyswarms.utils.plotters* module implements various visualization capabilities to interact with your swarm. Here, ou can plot cost history and animate your swarm in both 2D or 3D spaces.

pyswarms.utils.plotters.plotters module

Plotting tool for Optimizer Analysis

This module is built on top of `matplotlib` to render quick and easy plots for your optimizer. It can plot the best cost for each iteration, and show animations of the particles in 2-D and 3-D space. Furthermore, because it has `matplotlib` running under the hood, the plots are easily customizable.

For example, if we want to plot the cost, simply run the optimizer, get the cost history from the optimizer instance, and pass it to the `plot_cost_history()` method

```
import pyswarms as ps
from pyswarms.utils.functions.single_obj import sphere
from pyswarms.utils.plotters import plot_cost_history

# Set up optimizer
options = {'c1':0.5, 'c2':0.3, 'w':0.9}
optimizer = ps.single.GlobalBestPSO(n_particles=10, dimensions=2,
                                    options=options)

# Obtain cost history from optimizer instance
cost_history = optimizer.cost_history

# Plot!
plot_cost_history(cost_history)
plt.show()
```

In case you want to plot the particle movement, it is important that either one of the `matplotlib` animation `Writers` is installed. These doesn't come out of the box for `pyswarms`, and must be installed separately. For example, in a Linux or Windows distribution, you can install `ffmpeg` as

```
>>> conda install -c conda-forge ffmpeg
```

Now, if you want to plot your particles in a 2-D environment, simply pass the position history of your swarm (obtainable from swarm instance):

```
import pyswarms as ps
from pyswarms.utils.functions.single_obj import sphere
from pyswarms.utils.plotters import plot_cost_history

# Set up optimizer
options = {'c1':0.5, 'c2':0.3, 'w':0.9}
optimizer = ps.single.GlobalBestPSO(n_particles=10, dimensions=2,
                                    options=options)

# Obtain pos history from optimizer instance
pos_history = optimizer.pos_history

# Plot!
plot_contour(pos_history)
```

You can also supply various arguments in this method: the indices of the specific dimensions to be used, the limits of the axes, and the interval/ speed of animation.

pyswarms.utils.plotters.plotters.**plot_contour**(*pos_history*, *canvas=None*, *title='Trajectory'*, *mark=None*, *designer=None*, *mesher=None*, *animator=None*, ***kwargs*)

Draw a 2D contour map for particle trajectories

Here, the space is represented as a flat plane. The contours indicate the elevation with respect to the objective function. This works best with 2-dimensional swarms with their fitness in z-space.

Parameters

- **pos_history** (*numpy.ndarray or list*) – Position history of the swarm with shape `(iteration, n_particles, dimensions)`
- **canvas** (tuple of `matplotlib.figure.Figure` and `matplotlib.axes.Axes` (default is `None`)) – The (figure, axis) where all the events will be draw. If `None` is supplied, then plot will be drawn to a fresh set of canvas.
- **title** (str (default is `'Trajectory'`)) – The title of the plotted graph.
- **mark** (tuple (default is `None`)) – Marks a particular point with a red crossmark. Useful for marking the optima.
- **designer** (pyswarms.utils.formatters.Designer (default is `None`)) – Designer class for custom attributes
- **mesher** (pyswarms.utils.formatters.Mesher (default is `None`)) – Mesher class for mesh plots
- **animator** (pyswarms.utils.formatters.Animator (default is `None`)) – Animator class for custom animation
- ****kwargs** (*dict*) – Keyword arguments that are passed as a keyword argument to `matplotlib.axes.Axes` plotting function

Returns The drawn animation that can be saved to mp4 or other third-party tools

Return type `matplotlib.animation.FuncAnimation`

pyswarms.utils.plotters.plotters.**plot_cost_history**(*cost_history*, *ax=None*, *title='Cost History'*, *designer=None*, ***kwargs*)

Create a simple line plot with the cost in the y-axis and the iteration at the x-axis

Parameters

- **cost_history** (*list or numpy.ndarray*) – Cost history of shape `(iters, )` or length `iters` where each element contains the cost for the given iteration.
- **ax** (`matplotlib.axes.Axes` (default is `None`)) – The axes where the plot is to be drawn. If `None` is passed, then the plot will be drawn to a new set of axes.
- **title** (str (default is `'Cost History'`)) – The title of the plotted graph.
- **designer** (pyswarms.utils.formatters.Designer (default is `None`)) – Designer class for custom attributes
- ****kwargs** (*dict*) – Keyword arguments that are passed as a keyword argument to `matplotlib.axes.Axes`

Returns The axes on which the plot was drawn.

Return type `matplotlib.axes._subplots.AxesSubplot`

pyswarms.utils.plotters.plotters.**plot_surface**(*pos_history*, *canvas=None*, *title='Trajectory'*, *designer=None*, *mesher=None*, *animator=None*, *mark=None*, ***kwargs*)

Plot a swarm's trajectory in 3D

This is useful for plotting the swarm's 2-dimensional position with respect to the objective function. The value in the z-axis is the fitness of the 2D particle when passed to the objective function. When preparing the position history, make sure that the:

- first column is the position in the x-axis,
- second column is the position in the y-axis; and
- third column is the fitness of the 2D particle

The *pyswarms.utils.plotters.formatters.Mesher* class provides a method that prepares this history given a 2D pos history from any optimizer.

```
import pyswarms as ps
from pyswarms.utils.functions.single_obj import sphere
from pyswarms.utils.plotters import plot_surface
from pyswarms.utils.plotters.formatters import Mesher

# Run optimizer
options = {'c1':0.5, 'c2':0.3, 'w':0.9}
optimizer = ps.single.GlobalBestPSO(n_particles=10, dimensions=2, options)

# Prepare position history
m = Mesher(func=sphere)
pos_history_3d = m.compute_history_3d(optimizer.pos_history)

# Plot!
plot_surface(pos_history_3d)
```

Parameters

- **pos_history** (*numpy.ndarray*) – Position history of the swarm with shape `(iteration, n_particles, 3)`
- **objective_func** (*callable*) – The objective function that takes a swarm of shape `(n_particles, 2)` and returns a fitness array of `(n_particles, )`
- **canvas** (tuple of `matplotlib.figure.Figure` and) –

:param matplotlib.axes.Axes (default is None): The (figure, axis) where all the events will be draw. If None is supplied, then plot will be drawn to a fresh set of canvas.

Parameters

- **title** (str (default is `'Trajectory'`)) – The title of the plotted graph.
- **mark** (tuple (default is `None`)) – Marks a particular point with a red crossmark. Useful for marking the optima.
- **designer** (pyswarms.utils.formatters.Designer (default is `None`)) – Designer class for custom attributes
- **mesher** (pyswarms.utils.formatters.Mesher (default is `None`)) – Mesher class for mesh plots
- **animator** (pyswarms.utils.formatters.Animator (default is `None`)) – Animator class for custom animation
- ****kwargs** (*dict*) – Keyword arguments that are passed as a keyword argument to `matplotlib.axes.Axes` plotting function

Returns The drawn animation that can be saved to mp4 or other third-party tools

Return type `matplotlib.animation.FuncAnimation`

pyswarms.utils.plotters.formatters module

Plot Formatters

This module implements helpful classes to format your plots or create meshes.

class pyswarms.utils.plotters.formatters.**Animator**(*interval: int = 80, repeat_delay=None, repeat: bool = True*)

Bases: object

Animator class for specifying animation behavior

You can use this class to modify options on how the animation will be run in the pyswarms.utils.plotters.plot_contour() and pyswarms.utils.plotters.plot_surface() methods.

```
from pyswarms.utils.plotters import plot_contour
from pyswarms.utils.plotters.formatters import Animator

# Do not repeat animation
my_animator = Animator(repeat=False)

# Assuming we already had an optimizer ready
plot_contour(pos_history, animator=my_animator)
```

interval
int (default is 80) – Sets the interval or speed into which the animation is played.

repeat_delay
int, float (default is None) – Sets the delay before repeating the animation again.

repeat
bool (default is True) – Pass False if you don't want to repeat the animation.

class pyswarms.utils.plotters.formatters.**Designer**(*figsize: tuple = (10, 8), title_fontsize='large', text_fontsize='medium', legend='Cost', label=['x-axis', 'y-axis', 'z-axis'], limits=[(-1, 1), (-1, 1), (-1, 1)], colormap=<matplotlib.colors.ListedColormap object>*)

Bases: object

Designer class for specifying a plot's formatting and design

You can use this class for specifying design-related customizations to your plot. This can be passed in various functions found in the *pyswarms.utils.plotters* module.

```
from pyswarms.utils.plotters import plot_cost_history
from pyswarms.utils.plotters.formatters import Designer

# Set title_fontsize into 20
my_designer = Designer(title_fontsize=20)

# Assuming we already had an optimizer ready
plot_cost_history(cost_history, designer=my_designer)
```

figsize
tuple (default is (10,8)) – Overall figure size.

title_fontsize
str, int, or float (default is large) – Size of the plot's title.

text_fontsize
str, int, or float (default is medium) – Size of the plot's labels and legend.

legend
str (default is `Cost`) – Label to show in the legend. For cost histories, it states the label of the line plot.

label
str, list, or tuple (default is `['x-axis', 'y-axis', 'z-axis']`) – Label to show in the x, y, or z-axis. For a 3D plot, please pass an iterable with three elements.

limits
list (default is `[(-1, 1), (-1, 1), (-1, 1)]`) – The x-, y-, z- limits of the axes. Pass an iterable with the number of elements representing the number of axes.

colormap
matplotlib.cm.Colormap (default is `cm.viridis`) – Colormap for contour plots

class pyswarms.utils.plotters.formatters.**Mesher**(*func, delta: float = 0.001, limits=[(-1, 1), (-1, 1)], levels: list = array([-2. , -1.93, -1.86, -1.79, -1.72, -1.65, -1.58, -1.51, -1.44, -1.37, -1.3 , -1.23, -1.16, -1.09, -1.02, -0.95, -0.88, -0.81, -0.74, -0.67, -0.6 , -0.53, -0.46, -0.39, -0.32, -0.25, -0.18, -0.11, -0.04, 0.03, 0.1 , 0.17, 0.24, 0.31, 0.38, 0.45, 0.52, 0.59, 0.66, 0.73, 0.8 , 0.87, 0.94, 1.01, 1.08, 1.15, 1.22, 1.29, 1.36, 1.43, 1.5 , 1.57, 1.64, 1.71, 1.78, 1.85, 1.92, 1.99]), alpha: float = 0.3*)

Bases: `object`

Mesher class for plotting contours of objective functions

This class enables drawing a surface plot of a given objective function. You can customize how this plot is drawn with this class. Pass an instance of this class to enable meshing.

```
from pyswarms.utils.plotters import plot_surface
from pyswarms.utils.plotters.formatters import Mesher
from pyswarms.utils.functions import single_obj as fx

# Use sphere function
my_mesher = Mesher(func=fx.sphere)

# Assuming we already had an optimizer ready
plot_surface(pos_history, mesher=my_mesher)
```

func
callable – Objective function to plot a surface of.

delta
float (default is `0.001`) – Number of steps when generating the surface plot

limits
list, tuple (default is `[(-1,1), (-1,1)]`) – The range, in each axis, where the mesh will be drawn.

levels
list or int (default is `np.arange(-2.0, 2.0, 0.070)`) – Levels on which the contours are shown. If `int` is passed, then *matplotlib* automatically computes for the level positions.

alpha
float (default is `0.3`) – Transparency of the surface plot

limits
list (default is `[(-1, 1), (-1, 1)]`) – The x-, y-, z- limits of the axes. Pass an iterable with the number of elements representing the number of axes.

compute_history_3d(*pos_history*)
Compute a 3D position matrix

The first two columns are the 2D position in the x and y axes respectively, while the third column is the fitness on that given position.

Parameters **pos_history** (*numpy.ndarray*) – Two-dimensional position matrix history of shape `(iterations, n_particles, 2)`

Returns 3D position matrix of shape `(iterations, n_particles, 3)`

Return type numpy.ndarray

1.15.4 pyswarms.utils.reporter package

class pyswarms.utils.reporter.reporter.**Reporter**(*log_path=None*, *config_path=None*, *logger=None*, *printer=None*)

Bases: `object`

A Reporter object that abstracts various logging capabilities

To set-up a Reporter, simply perform the following tasks:

```
from pyswarms.utils import Reporter

rep = Reporter()
rep.log("Here's my message", lvl=logging.INFO)
```

This will set-up a reporter with a default configuration that logs to a file, *report.log*, on the current working directory. You can change the log path by passing a string to the *log_path* parameter:

```
from pyswarms.utils import Reporter

rep = Reporter(log_path="/path/to/log/file.log")
rep.log("Here's my message", lvl=logging.INFO)
```

If you are working on a module and you have an existing logger, you can pass that logger instance during initialization:

```
# mymodule.py
from pyswarms.utils import Reporter

# An existing logger in a module
logger = logging.getLogger(__name__)
rep = Reporter(logger=logger)
```

Lastly, if you have your own logger configuration (YAML file), then simply pass that to the *config_path* parameter. This overrides the default configuration (including *log_path*):

```
from pyswarms.utils import Reporter

rep = Reporter(config_path="/path/to/config/file.yml")
rep.log("Here's my message", lvl=logging.INFO)
```

__init__(*log_path=None*, *config_path=None*, *logger=None*, *printer=None*)
Initialize the reporter

log_path
str (default is `None`) – Sets the default log path (overriden when `path` is given to `_setup_logger()`)

config_path
str (default is `None`) – Sets the configuration path for custom loggers

logger
logging.Logger (default is `None`) – The logger object. By default, it creates a new `Logger` instance

printer
pprint.PrettyPrinter (default is `None`) – A printer object. By default, it creates a `PrettyPrinter` instance with default values

_load_defaults()
Load default logging configuration

_setup_logger(*path=None*)
Set-up the logger with default values

This method is called right after initializing the Reporter module. If no path is supplied, then it loads a default configuration. You can view the defaults via the Reporter._default_config attribute.

Parameters **path** (*str*) – Path to a YAML configuration. If not supplied, uses a default config.

hook(**args*, ***kwargs*)
Set a hook on the progress bar

Method for creating a postfix in tqdm. In practice we use this to report the best cost found during an iteration:

```
from pyswarms.utils import Reporter

rep = Reporter()
# Create a progress bar
for i in rep.pbar(100, name="Optimizer")
        best_cost = compute()
        rep.hook(best_cost=best_cost)
```

log(*msg*, *lvl=20*, **args*, ***kwargs*)
Log a message within a set level

This method abstracts the logging.Logger.log() method. We use this method during major state changes, errors, or critical events during the optimization run.

You can check logging levels on this **'link'**_. In essence, DEBUG is 10, INFO is 20, WARNING is 30, ERROR is 40, and CRITICAL is 50.

Parameters

- **msg** (*str*) – Message to be logged
- **lvl** (*int (default is 20)*) – Logging level

pbar(*iters*, *desc=None*)
Create a tqdm iterable

You can use this method to create progress bars. It uses a set of abstracted methods from tqdm:

```
from pyswarms.utils import Reporter

rep = Reporter()
# Create a progress bar
for i in rep.pbar(100, name="Optimizer")
        pass
```

Parameters

- **iters** (*int*) – Maximum range passed to the tqdm instance
- **desc** (*str*) – Name of the progress bar that will be displayed

Returns A tqdm iterable

Return type tqdm._tqdm.tqdm

print (*msg*, *verbosity*, *threshold=0*)
Print a message into console

This method can be called during non-system calls or minor state changes. In practice, we call this method when reporting the cost on a given timestep.

Parameters

- **msg** (*str*) – Message to be printed
- **verbosity** (*int*) – Verbosity parameter, prints message when it's greater than the threshold
- **threshold** (*int (default is 0)*) – Threshold parameer, prints message when it's lesser than the verbosity

1.15.5 pyswarms.utils.search package

The *pyswarms.utils.search* module implements various techniques in hyperparameter value optimization.

pyswarms.utils.search.base_search module

Base class for hyperparameter optimization search functions

class pyswarms.utils.search.base_search.**SearchBase** (*optimizer*, *n_particles*, *dimensions*, *options*, *objective_func*, *iters*, *bounds=None*, *velocity_clamp=(0, 1)*)

Bases: object

__init__ (*optimizer*, *n_particles*, *dimensions*, *options*, *objective_func*, *iters*, *bounds=None*, *velocity_clamp=(0, 1)*)
Initialize the Search

optimizer
pyswarms.single – either LocalBestPSO or GlobalBestPSO

n_particles
int – number of particles in the swarm.

dimensions
int – number of dimensions in the space.

options
dict with keys {'c1', 'c2', 'w', 'k', 'p'} – a dictionary containing the parameters for the specific optimization technique
- **c1** [float] cognitive parameter
- **c2** [float] social parameter
- **w** [float] inertia parameter
- **k** [int] number of neighbors to be considered. Must be a positive integer less than n_particles
- **p: int {1,2}** the Minkowski p-norm to use. 1 is the sum-of-absolute values (or L1 distance) while 2 is the Euclidean (or L2) distance.

objective_func
function – objective function to be evaluated

iters
int – number of iterations

bounds
tuple of np.ndarray, optional (default is None) – a tuple of size 2 where the first entry is the minimum bound while the second entry is the maximum bound. Each array must be of shape `(dimensions,)`.

velocity_clamp
tuple (default is `None`) – a tuple of size 2 where the first entry is the minimum velocity and the second entry is the maximum velocity. It sets the limits for velocity clamping.

assertions()
Assertion method to check `optimizer` input

Raises `TypeError` – When `optimizer` does not have an *'optimize'* attribute.

generate_score(*options*)
Generate score for optimizer's performance on objective function

Parameters **options** (*dict*) – a dict with the following keys: {'c1', 'c2', 'w', 'k', 'p'}

search(*maximum=False*)
Compare optimizer's objective function performance scores for all combinations of provided parameters

Parameters **maximum** (*bool*) – a bool defaulting to False, returning the minimum value for the objective function. If set to True, will return the maximum value for the objective function.

pyswarms.utils.search.grid_search module

Hyperparameter grid search.

Compares the relative performance of hyperparameter value combinations in optimizing a specified objective function.

For each hyperparameter, user can provide either a single value or a list of possible values. The cartesian products of these hyperparameters are taken to produce a grid of all possible combinations. These combinations are then tested to produce a list of objective function scores. The search method default returns the minimum objective function score and hyperparameters that yield the minimum score, yet maximum score can also be evaluated.

```
>>> options = {'c1': [1, 2, 3],
               'c2': [1, 2, 3],
               'w' : [2, 3, 5],
               'k' : [5, 10, 15],
               'p' : 1}
>>> g = GridSearch(LocalBestPSO, n_particles=40, dimensions=20,
                   options=options, objective_func=sphere, iters=10)
>>> best_score, best_options = g.search()
>>> best_score
0.498641604188
>>> best_options['c1']
1
>>> best_options['c2']
1
```

class pyswarms.utils.search.grid_search.**GridSearch**(*optimizer*, *n_particles*, *dimensions*, *options*, *objective_func*, *iters*, *bounds=None*, *velocity_clamp=(0, 1)*)

Bases: *pyswarms.utils.search.base_search.SearchBase*

Exhaustive search of optimal performance on selected objective function over all combinations of specified hyperparameter values.

__init__(*optimizer*, *n_particles*, *dimensions*, *options*, *objective_func*, *iters*, *bounds=None*, *velocity_clamp=(0, 1)*)
Initialize the Search

generate_grid()
Generate the grid of all hyperparameter value combinations

pyswarms.utils.search.random_search module

Hyperparameter random search.

Compares the relative performance of combinations of randomly generated hyperparameter values in optimizing a specified objective function.

User provides lists of bounds for the uniform random value generation of 'c1', 'c2', and 'w', and the random integer value generation of 'k'. Combinations of values are generated for the number of iterations specified, and the generated grid of combinations is used in the search method to find the optimal parameters for the objective function. The search method default returns the minimum objective function score and hyperparameters that yield the minimum score, yet maximum score can also be evaluated.

```
>>> options = {'c1': [1, 5],
               'c2': [6, 10],
               'w' : [2, 5],
               'k' : [11, 15],
               'p' : 1}
>>> g = RandomSearch(LocalBestPSO, n_particles=40, dimensions=20,
                     options=options, objective_func=sphere, iters=10)
>>> best_score, best_options = g.search()
>>> best_score
1.41978545901
>>> best_options['c1']
1.543556887693
>>> best_options['c2']
9.504769054771
```

class pyswarms.utils.search.random_search.**RandomSearch**(*optimizer*, *n_particles*, *dimensions*, *options*, *objective_func*, *iters*, *n_selection_iters*, *bounds=None*, *velocity_clamp=(0, 1)*)
Bases: *pyswarms.utils.search.base_search.SearchBase*

Search of optimal performance on selected objective function over combinations of randomly selected hyperparameter values within specified bounds for specified number of selection iterations.

__init__(*optimizer*, *n_particles*, *dimensions*, *options*, *objective_func*, *iters*, *n_selection_iters*, *bounds=None*, *velocity_clamp=(0, 1)*)
Initialize the Search

n_selection_iters
int – number of iterations of random parameter selection

assertions()
Assertion method to check `n_selection_iters` input

Raises `TypeError` – When `n_selection_iters` is not of type int

generate_grid()
Generate the grid of hyperparameter value combinations

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Bibliography

[CI2007] 1. Engelbrecht, “An Introduction to Computational Intelligence,” John Wiley & Sons, 2007.

[IJCNN1995] 10. Kennedy and R.C. Eberhart, “Particle Swarm Optimization,” Proceedings of the IEEE International Joint Conference on Neural Networks, 1995, pp. 1942-1948.

[ICEC2008] 25. Shi and R.C. Eberhart, “A modified particle swarm optimizer,” Proceedings of the IEEE International Conference on Evolutionary Computation, 1998, pp. 69-73.

[IJCNN1995] J. Kennedy and R.C. Eberhart, “Particle Swarm Optimization,” Proceedings of the IEEE International Joint Conference on Neural Networks, 1995, pp. 1942-1948.

[IJCNN1995] J. Kennedy and R.C. Eberhart, “Particle Swarm Optimization,” Proceedings of the IEEE International Joint Conference on Neural Networks, 1995, pp. 1942-1948.

[MHS1995] J. Kennedy and R.C. Eberhart, “A New Optimizer using Particle Swarm Theory,” in Proceedings of the Sixth International Symposium on Micromachine and Human Science, 1995, pp. 39–43.

[IJCNN1995] J. Kennedy and R.C. Eberhart, “Particle Swarm Optimization,” Proceedings of the IEEE International Joint Conference on Neural Networks, 1995, pp. 1942-1948.

[SMC1997] J. Kennedy and R.C. Eberhart, “A discrete binary version of particle swarm algorithm,” Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics, 1997.

Python Module Index

p

Index

Symbols

A

B

C

D

E

F

P

R

S

T

U

V

Z