PHYSBO Documentation

Release 1.0.1

PHYSBO's team

May 25, 2022

CONTENTS:

CHAPTER

ONE

INTRODUCTION

1.1 About PHYSBO

PHYSBO (optimization tools for PHYSics based on Bayesian Optimization) is a Python library for fast and scalable Bayesian optimization. It is based on COMBO (Common Bayesian Optimization) and has been developed mainly for researchers in the materials science field. There are many attempts to accelerate scientific discovery through data-driven design-of-experiment algorithms in the fields of physics, chemistry, and materials. Bayesian optimization is an effective tool for accelerating these scientific discoveries. Bayesian optimization is a technique that can be used for complex simulations and real-world experimental tasks where the evaluation of objective function values (e.g., characteristic values) is very costly. In other words, the problem solved by Bayesian optimization is to find a parameter (e.g., material composition, structure, process and simulation parameters) with a better objective function value (e.g., material properties) in as few experiments and simulations as possible. In Bayesian optimization, the candidate parameters to be searched for are listed in advance, and the candidate with the largest objective function value is selected from among the candidates by making good use of machine learning (using Gaussian process regression) prediction. Experiments and simulations are performed on the candidates and the objective function values are evaluated. By repeating the process of selection by machine learning and evaluation by experimental simulation, we can reduce the number of times of optimization. On the other hand, Bayesian optimization is generally computationally expensive, and standard implementations such as scikit-learn are difficult to handle a large amount of data. PHYSBO achieves high scalability due to the following features

- Thompson Sampling
- random feature map
- one-rank Cholesky update
- automatic hyperparameter tuning

Please see this reference for technical details.

1.2 Citation

When citing PHYSBO, please cite the following reference:

Yuichi Motoyama, Ryo Tamura, Kazuyoshi Yoshimi, Kei Terayama, Tsuyoshi Ueno, Koji Tsuda, Bayesian optimization package: PHYSBO, Computer Physics Communications Volume 278, September 2022, 108405. Available from https://www.sciencedirect.com/science/article/pii/S0010465522001242?via%3Dihub (open access).

Bibtex is given as follows:

```
@misc{@article{MOTOYAMA2022108405,
title = {Bayesian optimization package: PHYSBO},
```

(continues on next page)

(continued from previous page)

```
journal = {Computer Physics Communications},
volume = {278},
pages = {108405},
year = {2022},
issn = {0010-4655},
doi = {https://doi.org/10.1016/j.cpc.2022.108405},
author = {Yuichi Motoyama and Ryo Tamura and Kazuyoshi Yoshimi and Kei Terayama and␣
↪Tsuyoshi Ueno and Koji Tsuda},
keywords = {Bayesian optimization, Multi-objective optimization, Materials screening,␣
↪Effective model estimation}
}
```

1.3 Main Developers

- ver. 1.0-
  - Ryo Tamura (International Center for Materials Nanoarchitectonics, National Institute for Materials Science)
  - Tsuyoshi Ueno (Magne-Max Capital Management Company)
  - Kei Terayama (Graduate School of Medical Life Science, Yokohama City University)
  - Koji Tsuda (Graduate School of Frontier Sciences, The University of Tokyo)
  - Yuichi Motoyama (The Institute for Solid State Physics, The University of Tokyo)
  - Kazuyoshi Yoshimi (The Institute for Solid State Physics, The University of Tokyo)
  - Naoki Kawashima (The Institute for Solid State Physics, The University of Tokyo)

1.4 License

Part of this software is developed under the support of "Project for advancement of software usability in materials science" by The Institute for Solid State Physics, The University of Tokyo.

CHAPTER

TWO

BASIC USAGE

2.1 Install

2.1.1 Required Packages

- Python >= 3.6
- numpy
- scipy

2.1.2 Download and Install

- From PyPI (recommended)

  ```
  $ pip3 install physbo
  ```

  - Required packages such as NumPy will also be installed at the same time.
  - If you add the `--user` option, it will be installed under the user's home directory

    ```
    $ pip3 install --user physbo
    ```

- From source (for developers)

  1. Download or clone the github repository

     $ git clone https://github.com/issp-center-dev/PHYSBO

  2. Update `pip` to 19.0 or higher

     ```
     $ pip3 install -U pip

     - If you don't have ``pip3``, you can install it with ``python3 -m ensurepip``.
     ```

  3. Install

     ```
     $ cd PHYSBO
     $ pip3 install --user ./
     ```

2.1.3 Uninstall

1. Execute the following command.

```
$ pip uninstall physbo
```

2.2 Basic structures

PHYSBO has the following structure (shown up to the second level).

Each module is created with the following structure.

- `blm` :Module for Baysean linear model
- `gp` :Module for Gaussian Process
- `opt` :Module for optimazation
- `search` :Module for searching for optimal solutions
- `predictor.py` :Abstract class for predictors
- `variable.py` :Class defined for variable associations used in physbo
- `misc` : Others (e.g., modules for normalizing the search space)

For more information about each module, please refer to the API reference.

2.3 Calculation flow

Bayesian optimization is well suited for optimization problems such as complex simulations or real-world experimental tasks where the objective function is very costly to evaluate. In PHYSBO, the following steps are used to perform the optimization (please refer to the tutorial and API reference for details on each).

1. Defining the search space

   Define each parameter set (d-dimensional vector) as a search candidate, where N: the number of search candidates , d: the number of input parameter dimensions. The parameter set should list all the candidates.

2. Defining the simulator

   For searching candidates defined above, define a simulator that gives the objective function values (values to be optimized, such as material property values) for each search candidate. In PHYSBO, the direction of optimization is to maximize the objective function, so if you want to minimize the objective function, you can do so by applying a negative value to the value returned by the simulator.

3. Performing optimization

   First, set the optimization policy (the search space is passed to policy as an argument at this stage). You can choose between the following two optimization methods.

   - `random_search`
   - `bayes_search`

   In `random_search`, we randomly select parameters from the search space and search for the largest objective function among them. It is used to prepare an initial set of parameters as a preprocessing step for Bayesian optimization. `bayes_search` performs Bayesian optimization. The type of score (acquisition function) in Bayesian optimization can be one of the following.

- TS (Thompson Sampling): Sample one regression function from the posterior probability distribution of the learned Gaussian process, and select the point where the predicetd value becomes maximum as a next candidate.
- EI (Expected Improvement): Select the point where the expected value of the difference between the predicted value by the Gaussian process and the maximum value in the current situation becomes the maximum as a next candidate.
- PI (Probability of Improvement): Select the point with the highest probability of exceeding the current maximum of the current acquisition function as a next candidate.

Details of Gaussian processes are described in *Algorithm* . For other details of each method, please see this reference . If you specify the simulator and the number of search steps in these methods, the following loop will rotate by the number of search steps.

i). Select the next parameter to be executed from the list of candidate parameters.

ii). Run the simulator with the selected parameters.

The number of parameter returned in i) is one by default, but it is possible to return multiple parameters in one step. For more details, please refer to the "Exploring multiple candidates at once" section of the tutorial. Also, instead of running the above loop inside PHYSBO, it is possible to control i) and ii) separately from the outside. In other words, it is possible to propose the next parameter to be executed from PHYSBO, evaluate its objective function value in some way outside PHYBO (e.g., by experiment rather than numerical calculation), and register the evaluated value in PHYSBO. For more details, please refer to the "Running Interactively" section of the tutorial.

4. Check numerical results

The search result `res` is returned as an object of the `history` class (`physbo.search.discrete.results.history`). The following is a reference to the search results.

- `res.fx`: The logs of evaluation values for simulator (objective function) simulator.
- `res.chosen_actions`: The logs of the action ID (parameter) when the simulator has executed.
- `fbest, best_action= res.export_all_sequence_best_fx()`: The logs of the best values and their action IDs (parameters) at each step where the simulator has executed.
- `res.total_num_search`: Total number steps where the simulator has executed.

The search results can be saved to an external file using the `save` method, and the output results can be loaded using the `load` method. See the tutorial for details on how to use it.

CHAPTER

THREE

TUTORIALS

Here, the usage of PHYSBO is introduced through tutorials.

3.1 Basic usage of PHYSBO

3.1.1 Introduction

In this tutorial, we will introduce how to define the simulator class and find the minimum value of a one-dimensional function using PHYSBO.

First, we will import PHYSBO.

[1]:
```
import physbo
```

3.1.2 Defining the search space

In the following example, the search space `X` is defined as a grid chopped by `window_num=10001` divisions from `x_min = -2.0` to `x_max = 2.0`. Note that `X` must be in `window_num` x `d` ndarray format (`d` is the number of dimensions, in this case one). In this case, `d` is the number of dimensions, in this case two, so we use reshape to transform it.

[2]:
```
#In
import numpy as np
import scipy
import physbo
import itertools

#In
#Create candidate
window_num=10001
x_max = 2.0
x_min = -2.0

X = np.linspace(x_min,x_max,window_num).reshape(window_num, 1)
```

3.1.3 Defining the simulator class

Here, we define the simulator class to set as the objective function.

In this case, the problem is to find the minimum x such that $f(x) = 3x^4 + 4x^3 + 1.0$ (the answer is $x = -1.0$).

In the simulator class, we define the `__call__` function (or `__init__` if there are initial variables, etc.). (If there are initial variables, define `__init__`.) The action indicates the index number of the grid to be retrieved from the search space, and is generally in the form of an ndarray so that multiple candidates can be calculated at once. In this case, we choose one candidate point from `X` as `action_idx=action[0]` to calculate only one candidate at a time. Since **PHYSBO is designed to find the maximum value of the objective function**, it returns the value of f(x) at the candidate point multiplied by -1.

[3]:
```python
# Declare the class for calling the simulator.
class simulator:

    def __call__(self, action ):
        action_idx = action[0]
        x = X[action_idx][0]
        fx = 3.0*x**4 + 4.0*x**3 + 1.0
        fx_list.append(fx)
        x_list.append(X[action_idx][0])

        print ("*********************")
        print ("Present optimum interactions")

        print ("x_opt=", x_list[np.argmin(np.array(fx_list))])

        return -fx
```

3.1.4 Performing optimization

Setting policy

First, set the optimization `policy`.

Next, set `test_X` to the matrix of search candidates (`numpy.array`).

[4]:
```python
# set policy
policy = physbo.search.discrete.policy(test_X=X)

# set seed
policy.set_seed(0)
```

When `policy` is set, no optimization is done yet. Execute the following methods on `policy` to optimize it.

- `random_search`.
- `bayes_search`.

If you specify the `simulator` and the number of search steps in these methods, the following loop will be executed for the number of search steps.

i) Select the next parameter to be executed from the candidate parameters.

ii) Execute `simulator` with the selected parameters.

The default number of parameter returned by i) is one, but it is possible to return multiple parameters in one step. See the section "Searching for multiple candidates at once" for details.

Also, instead of running the above loop inside PHYSBO, it is possible to control i) and ii) separately from the outside. In other words, it is possible to propose the next parameter to be executed from PHYSBO, evaluate its objective function value in some way outside PHYBO (e.g., by experiment rather than numerical calculation), propose it in some way outside PHYSBO, and register the evaluated value in PHYSBO. For more details, please refer to the "Running Interactively" section of the tutorial.

Random Search

First of all, let's perform a random search.

Since Bayesian optimization requires at least two objective function values to be obtained (the initial number of data required depends on the problem to be optimized and the dimension d of the parameters), we will first perform a random search.

argument.

- `max_num_probes`: Number of search steps.
- `simulator`: The simulator of the objective function (an object of class simulator).

[ ]:
```
fx_list=[]
x_list = []

res = policy.random_search(max_num_probes=20, simulator=simulator())
```

When executed, the objective function value and its action ID for each step, and the best value up to now and its action ID will be printed as follows.

```
0020-th step: f(x) = -19.075990 (action=8288)
   current best f(x) = -0.150313 (best action=2949)
```

Bayesian Optimization

Next, we run the Bayesian optimization as follows.

argument.

- `max_num_probes`: Number of search steps.
- `simulator`: The simulator of the objective function (an object of class simulator).
- `score`: The type of acquisition function. You can specify one of the following
  - TS (Thompson Sampling)
  - EI (Expected Improvement)
  - PI (Probability of Improvement)
- `interval`: The hyperparameters are trained at the specified interval. If a negative value is specified, no hyperparameter will be learned. 0 means that hyperparameter learning will be performed only in the first step.
- `num_rand_basis`: Number of basis functions. 0 means that a normal Gaussian process without Bayesian linear model will be used.

```
[ ]: res = policy.bayes_search(max_num_probes=50, simulator=simulator(), score='TS',
                                                  interval=0, num_rand_basis=500)
```

3.1.5 Checking the results

The search result `res` is returned as an object of the `history` class (`physbo.search.discrete.results.history`).
The following is a reference to the search results.

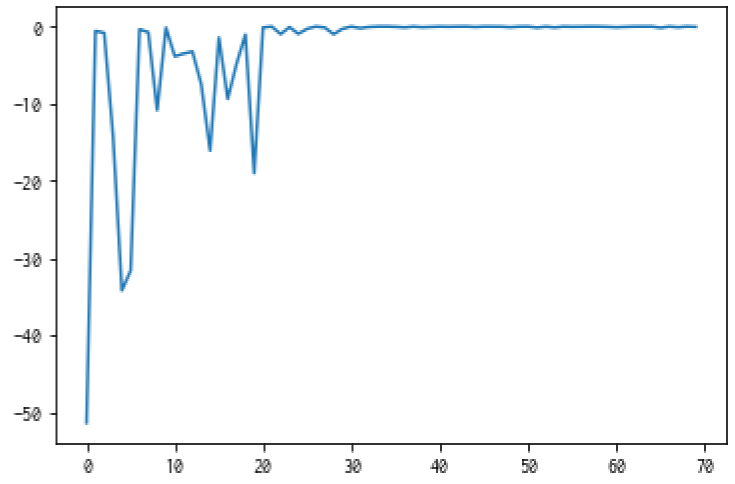
- `res.fx` : The history of evaluated values of simulator (objective function).
- `res.chosen_actions`: The history of action IDs (parameters) when the simulator was evaluated.
- `fbest, best_action= res.export_all_sequence_best_fx()`: The history of best values and their action IDs (parameters) for all timings when the simulator was evaluated.
- `res.total_num_search`: Total number of simulator evaluations.

Let's plot the objective function value and the best value at each step.
`res.fx` and `best_fx` should range up to `res.total_num_search`, respectively.

```
[7]: import matplotlib.pyplot as plt
     %matplotlib inline
```

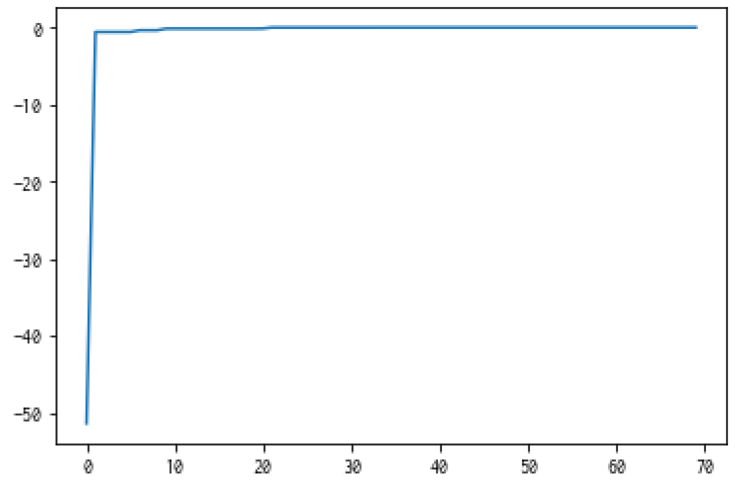
```
[8]: plt.plot(res.fx[0:res.total_num_search])
```

```
[8]: [<matplotlib.lines.Line2D at 0x7f816129d820>]
```



```
[9]: best_fx, best_action = res.export_all_sequence_best_fx()
     plt.plot(best_fx)
```

```
[9]: [<matplotlib.lines.Line2D at 0x7f8130b30f70>]
```



3.1.6 Serializing the results

The search results can be saved to an external file using the `save` method.

[10]:
```
res.save('search_result.npz')
```

[11]:
```
del res
```

Load the saved result file as follows:

[12]:
```
res = physbo.search.discrete.results.history()
res.load('search_result.npz')
```

Finally, the candidate with the best score can be displayed as follows. You can see that we have arrived at the correct solution $x = -1$.

[13]:
```
print(X[int(best_action[-1])])
```

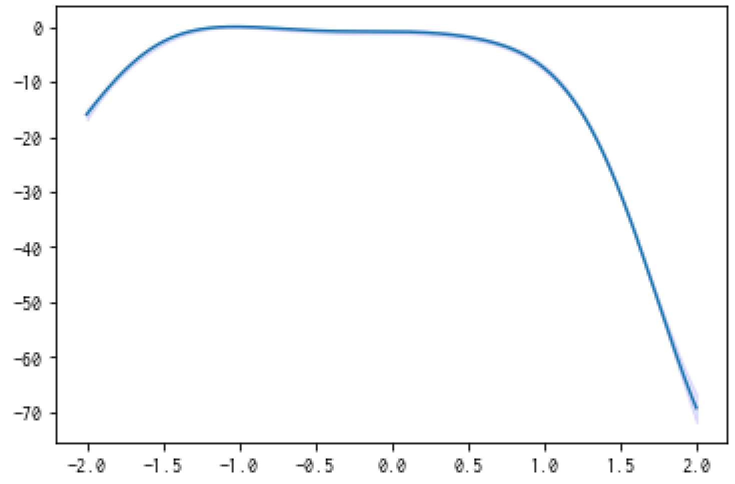
```
[-1.002]
```

3.1.7 Prediction

`policy` serves `get_post_fmean` and `get_post_fcov` methods for calculating mean value and variance of Gaussian process (post distribution).

[14]:
```
mean = policy.get_post_fmean(X)
var = policy.get_post_fcov(X)
std = np.sqrt(var)
xs = X[:,0]

ax = plt.subplot()
ax.plot(xs, mean)
ax.fill_between(xs, mean-std, mean+std, color="blue", alpha=.1)
```

```
[14]: <matplotlib.collections.PolyCollection at 0x7f81613d8370>
```

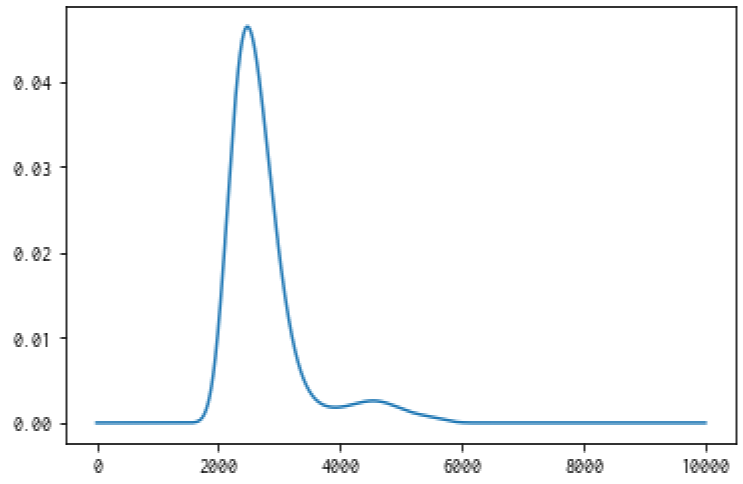


3.1.8 Acquisition function

`policy` serves `get_score` method for calculating acquisition function.

```
[15]: scores = policy.get_score(mode="EI", xs=X)
      plt.plot(scores)
```

```
[15]: [<matplotlib.lines.Line2D at 0x7f8130b84070>]
```



3.1.9 Parallelization

PHYSBO can calculate acquisition functions for candidates in parallel by using MPI via `mpi4py` . To enable MPI parallelization, pass a MPI communicator such as `MPI.COMM_WORLD` to a keyword argument, `comm` of the constructor of the `policy`.

[16]:
```
# from mpi4py import MPI
# policy = physbo.search.discrete.policy(test_X=X, comm=MPI.COMM_WORLD)
```

3.2 Gaussian process

PHYSBO performs Bayesian optimization while running Gaussian process regression.

Therefore, it is possible to run Gaussian process regression given training data, and to predict test data using the trained model.

In this section, the procedure is introduced.

3.2.1 Preparation of search candidate data

In this tutorial, the dataset for s5-210.csvthe problem of finding a stable interface structure for Cu is used as an example. The values that have already been evaluated are used, although the evaluation of the objective function, i.e., the structural relaxation calculation, actually takes on the order of several hours per calculation. For more information on the problem setup, please refer to the following references

- S. Kiyohara, H. Oda, K. Tsuda and T. Mizoguchi, "Acceleration of stable interface structure searching using a kriging approach", Jpn. J. Appl. Phys. 55, 045502 (2016).

Save the dataset file s5-210.csv into the subdirectory `data`, and load dataset from this file as the following:

[1]:
```
import physbo

import numpy as np

def load_data():
    A =  np.asarray(np.loadtxt('data/s5-210.csv',skiprows=1, delimiter=',') )
    X = A[:,0:3]
    t  = -A[:,3]
    return X, t

X, t = load_data()
X = physbo.misc.centering( X )
```

3.2.2 Defining training data

A randomly selected 10% of the target data will be used as training data, and another randomly selected 10% will be used as test data.

[2]:
```
N = len(t)
Ntrain = int(N*0.1)
Ntest = min(int(N*0.1), N-Ntrain)

id_all   = np.random.choice(N, N, replace=False)
id_train  = id_all[0:Ntrain]
id_test = id_all[Ntrain:Ntrain+Ntest]

X_train = X[id_train]
X_test = X[id_test]

t_train = t[id_train]
t_test = t[id_test]

print("Ntrain =", Ntrain)
print("Ntest =", Ntest)
```

```
Ntrain = 1798
Ntest = 1798
```

3.2.3 Learning and Prediction of Gaussian Processes

The following process is used to learn the Gaussian process and predict the test data.

1. Generate a model of the Gaussian process
2. The model is trained using X_train (parameters of the training data) and t_train (objective function value of the training data).
3. Run predictions on the test data (X_test) using the trained model.

Definition of covariance (Gaussian)

[3]:
```
cov = physbo.gp.cov.gauss( X_train.shape[1],ard = False )
```

Definition of mean value

[4]:
```
mean = physbo.gp.mean.const()
```

Definition of likelihood function (Gaussian)

[5]:
```
lik = physbo.gp.lik.gauss()
```

Generation of a Gaussian Process Model

[6]:
```
gp = physbo.gp.model(lik=lik,mean=mean,cov=cov)
config = physbo.misc.set_config()
```

Learning a Gaussian process model.

[7]:
```
gp.fit(X_train, t_train, config)
```

```
Start the initial hyper parameter searching ...
Done

Start the hyper parameter learning ...
0 -th epoch marginal likelihood 17312.31220145003
50 -th epoch marginal likelihood 6291.292745798703
100 -th epoch marginal likelihood 3269.1167759139516
150 -th epoch marginal likelihood 1568.3930580794922
200 -th epoch marginal likelihood 664.2847129159145
250 -th epoch marginal likelihood -249.28468708456558
300 -th epoch marginal likelihood -869.7604930929888
350 -th epoch marginal likelihood -1316.6809532065581
400 -th epoch marginal likelihood -1546.1623851368954
450 -th epoch marginal likelihood -1660.7298135295766
500 -th epoch marginal likelihood -1719.5056128528097
Done
```

Output the parameters in the learned Gaussian process.

[8]:
```
gp.print_params()
```

```


likelihood parameter =   [-2.81666924]
mean parameter in GP prior:  [-1.05939674]
covariance parameter in GP prior:  [-0.91578975 -2.45544347]
```

Calculating the mean (predicted value) and variance of the test data

[9]:
```
gp.prepare(X_train, t_train)
fmean = gp.get_post_fmean(X_train, X_test)
fcov = gp.get_post_fcov(X_train, X_test)
```

Results of prediction

[10]:
```
fmean
```

[10]:
```
array([-1.00420815, -1.10923758, -0.97840623, ..., -1.00323733,
       -0.97015759, -1.11076236])
```

Results of covariance

[11]:
```
fcov
```

[11]:
```
array([0.00056069, 0.00075529, 0.00043006, ..., 0.0016925 , 0.00070103,
       0.00073499])
```

Output mean square error of prediction

[12]:
```
np.mean((fmean-t_test)**2)
```

[12]:
```
0.008107085662147708
```

3.2.4 Prediction by trained models

Read the parameters of the trained model as gp_params and make predictions using them.

By storing gp_params and training data (X_train, t_train), prediction by the trained model is possible.

Prepare the learned parameters (must be done immediately after learning)

[13]:
```
#Prepare the learned parameters as a 1D array
gp_params =  np.append(np.append(gp.lik.params, gp.prior.mean.params), gp.prior.cov.
↪params)

gp_params
```

[13]:
```
array([-2.81666924, -1.05939674, -0.91578975, -2.45544347])
```

Prepare a model similar to the one used for training as gp

[14]:
```
#Definition of covariance (Gaussian)
cov = physbo.gp.cov.gauss( X_train.shape[1],ard = False )

#Definition of mean value
mean = physbo.gp.mean.const()

#Definition of likelihood function (Gaussian)
lik = physbo.gp.lik.gauss()

#Generation of a Gaussian Process Model
gp = physbo.gp.model(lik=lik,mean=mean,cov=cov)
```

Prepare a model similar to the one used for training as gp

[15]:
```
#Input learned parameters into the Gaussian process.
gp.set_params(gp_params)


#Calculate the mean (predicted value) and variance of the test data
gp.prepare(X_train, t_train)
fmean = gp.get_post_fmean(X_train, X_test)
fcov = gp.get_post_fcov(X_train, X_test)
```

Results of prediction

[16]:
```
fmean
```

[16]:
```
array([-1.00420815, -1.10923758, -0.97840623, ..., -1.00323733,
       -0.97015759, -1.11076236])
```

Results of covariance

[17]:
```
fcov
```

```
[17]: array([0.00056069, 0.00075529, 0.00043006, ..., 0.0016925 , 0.00070103,
             0.00073499])
```

Output mean square error of prediction

```
[18]: np.mean((fmean-t_test)**2)
```

```
[18]: 0.008107085662147708
```

```
[ ]:
```

3.3 Running PHYSBO interactively

You can run PHYSBO interactively in the following way:

1. Get the next parameter to run from PHYSBO
2. Get the evaluation values outside of PHYSBO
3. Register the evaluation values into PHYSBO

For example, it is suitable for the following cases.

- You want to perform an experiment manually and give the evaluation values to PHYSBO.
- You want to control the execution flexibly, such as running the simulator in a separate process.

3.3.1 Preparation of search candidate data

As the previous tutorials, save the dataset file s5-210.csv into the subdirectory `data`, and load dataset from this file as the following:

```
[1]: import physbo

import numpy as np


def load_data():
    A =  np.asarray(np.loadtxt('data/s5-210.csv',skiprows=1, delimiter=',') )
    X = A[:,0:3]
    t  = -A[:,3]
    return X, t

X, t = load_data()
X = physbo.misc.centering(X)
```

3.3.2 Definition of simulator

[2]:
```
class simulator:
    def __init__( self ):
        _, self.t = load_data()

    def __call__( self, action ):
        return self.t[action]
```

3.3.3 Executing optimization

[3]:
```
# Set policy
policy = physbo.search.discrete.policy(test_X=X)

# Set seed
policy.set_seed( 0 )
```

In each search step, the following processes are performed.

1. Running random_search or bayes_search with `max_num_probes=1, simulator=None` to get action IDs (parameters).
2. Getting the evaluation value (array of actions) by `t = simulator(actions)`.
3. Registering the evaluation value for the action ID (parameter) with `policy.write(actions, t)`.
4. Showing the history with `physbo.search.utility.show_search_results`.

In the following, we will perform two random sampling (1st, and 2nd steps) and two Bayesian optimization proposals (3rd, and 4th steps).

[ ]:
```
simulator = simulator()

''' 1st step (random sampling) '''
actions = policy.random_search(max_num_probes=1, simulator=None)
t  = simulator(actions)
policy.write(actions, t)
physbo.search.utility.show_search_results(policy.history, 10)

''' 2nd step (random sampling) '''
actions = policy.random_search(max_num_probes=1, simulator=None)
t = simulator(actions)
policy.write(actions, t)
physbo.search.utility.show_search_results(policy.history, 10)

''' 3rd step (bayesian optimization) '''
actions = policy.bayes_search(max_num_probes=1, simulator=None, score='EI', interval=0, ␣
↪num_rand_basis = 5000)
t = simulator(actions)
policy.write(actions, t)
physbo.search.utility.show_search_results(policy.history, 10)

''' 4-th step (bayesian optimization) '''
actions = policy.bayes_search(max_num_probes=1, simulator=None, score='EI', interval=0, ␣
↪num_rand_basis = 5000)
```

(continues on next page)

(continued from previous page)

```
t = simulator(actions)
policy.write(actions, t)
physbo.search.utility.show_search_results(policy.history, 10)
```

3.3.4 Suspend and restart

You can suspend and restart the optimization process by saving the following predictor, training, and history to an external file.

- predictor: Prediction model of the objective function
- training: Data used to train the predictor (`physbo.variable` object)
- history: History of optimization runs (`physbo.search.discrete.results.history` object)

```
[5]: policy.save(file_history='history.npz', file_training='training.npz', file_predictor=
     ↪'predictor.dump')
```

```
[ ]: # delete policy
     del policy

     # load policy
     policy = physbo.search.discrete.policy(test_X=X)
     policy.load(file_history='history.npz', file_training='training.npz', file_predictor=
     ↪'predictor.dump')

     ''' 5-th step (bayesian optimization) '''
     actions = policy.bayes_search(max_num_probes=1, simulator=None, score='EI', interval=0, ␣
     ↪num_rand_basis = 5000)
     t = simulator(actions)
     policy.write(actions, t)
     physbo.search.utility.show_search_results(policy.history, 10)

     # It is also possible to specify predictor and training separately.
     ''' 6-th step (bayesian optimization) '''
     actions = policy.bayes_search(max_num_probes=1,
                                               predictor=policy.predictor, training=policy.
     ↪training,
                                               simulator=None, score='EI', interval=0,  num_
     ↪rand_basis = 5000)
     t = simulator(actions)
     policy.write(actions, t)
     physbo.search.utility.show_search_results(policy.history, 10)
```

```
[ ]:
```

3.4 Restart calculations by reading existing calculation results

You can read existing action IDs (parameters) and their evaluation values and run PHYSBO in the following flow.

1. Load an external file and read the existing action IDs (parameters) and their evaluation values.
2. Register the action ID (parameter) and evaluation value to PHYSBO.
3. Get the parameters for the next execution from PHYSBO.

This can be used in cases where PHYSBO cannot be left open for a long time due to time constraints, and thus cannot be executed interactively.

3.4.1 Prepare the search candidate data

As the previous tutorials, save the dataset file s5-210.csv into the subdirectory `data`, and load dataset from this file as the following:

```
[2]: import physbo

import numpy as np


def load_data():
    A =  np.asarray(np.loadtxt('data/s5-210.csv',skiprows=1, delimiter=',') )
    X = A[:,0:3]
    t  = -A[:,3]
    return X, t

X, t = load_data()
X = physbo.misc.centering(X)
```

3.4.2 Preparing the precomputed data

In the `load_data` function above, all X and t are stored. Here, as precomputed, we get a random list of 20 actoin IDs and their evaluation values.

```
[4]: import random
random.seed(0)
calculated_ids = random.sample(range(t.size), 20)
print(calculated_ids)
t_initial = t[calculated_ids]
```

```
[12623, 13781, 1326, 8484, 16753, 15922, 13268, 9938, 15617, 11732, 7157, 16537, 4563,␣
↪9235, 4579, 3107, 8208, 17451, 4815, 10162]
```

3.4.3 Register action ID (parameter) and evaluation value to PHYSBO.

Register `calculated_ids` and `t[calculated_ids]` as a list in the initial variable `initial_data` of policy.

```
[5]: # set policy
policy = physbo.search.discrete.policy(test_X=X, initial_data=[calculated_ids, t_
↪initial])

# set seed
policy.set_seed( 0 )
```

3.4.4 Get the next parameter to be executed from PHYSBO

Perform Bayesian optimization to obtain the next candidate point.

```
[6]: actions = policy.bayes_search(max_num_probes=1, simulator=None, score="TS", interval=0, ␣
↪num_rand_basis = 5000)
print(actions, X[actions])
```

```
Start the initial hyper parameter searching ...
Done

Start the hyper parameter learning ...
0 -th epoch marginal likelihood -20.09302189053099
50 -th epoch marginal likelihood -23.11964735598211
100 -th epoch marginal likelihood -24.83020118385076
150 -th epoch marginal likelihood -25.817906570042602
200 -th epoch marginal likelihood -26.42342027124426
250 -th epoch marginal likelihood -26.822598600211865
300 -th epoch marginal likelihood -27.10872736571494
350 -th epoch marginal likelihood -27.331572599126865
400 -th epoch marginal likelihood -27.517235815448124
450 -th epoch marginal likelihood -27.67892333553869
500 -th epoch marginal likelihood -27.82299469827059
Done

[73] [[-1.6680279  -1.46385011  1.68585446]]
```

Perform external calculations on the obtained candidate points, and register the actions and their scores in a file. The process of reading the file again, running the Bayesian optimization, and obtaining the next candidate point is repeated to advance the Bayesian optimization.

3.5 Search multiple candidates at once

This is a tutorial for evaluating two or more candidates at once in a single search step.

3.5.1 Prepare the search candidate data

As the previous tutorials, save the dataset file s5-210.csv into the subdirectory `data`, and load dataset from this file as the following:

[1]:
```
import physbo

import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline


def load_data():
    A =  np.asarray(np.loadtxt('data/s5-210.csv',skiprows=1, delimiter=',') )
    X = A[:,0:3]
    t  = -A[:,3]
    return X, t

X, t = load_data()
X = physbo.misc.centering(X)
```

3.5.2 Definition of simulator

If `num_search_each_probe` (described below) is set to 2 or more, action will be input as an array of action IDs. Thus, define the simulator to return a list of evaluation values for each action ID.

The definitions in the basic tutorial and simulator are the same, but keep in mind that t is a `numpy.array`, and when action is an array, `self.t[action]` will also be an array.

[2]:
```
class simulator:
    def __init__( self ):
        _, self.t = load_data()

    def __call__( self, action ):
        return self.t[action]
```

Example of running the simulator

[3]:
```
sim = simulator()
sim([1,12,123])
```

[3]:
```
array([-1.01487066, -1.22884748, -1.05572838])
```

3.5.3 Performing optimizations

```
[4]: # set policy
policy = physbo.search.discrete.policy(test_X=X)

# set seed
policy.set_seed( 0 )
```

num_search_each_probe allows you to specify the number of candidates to evaluate in each search step.

In the following example, the simulator will be evaluated 2 × 10 = 20 times by random search and 8 × 10 = 80 times by Bayesian optimization.

argument.

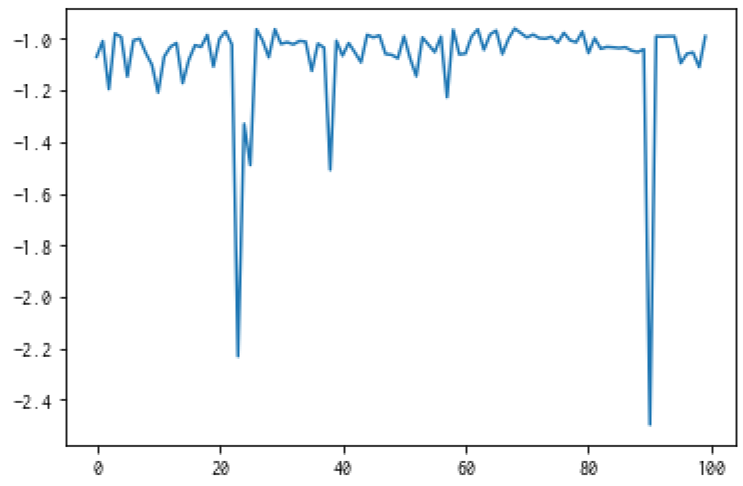
- max_num_probes: Number of search steps.
- num_search_each_probe: Number of candidates to evaluate at each search step.

```
[ ]: res = policy.random_search(max_num_probes=2, num_search_each_probe=10,␣
↪simulator=simulator())

res = policy.bayes_search(max_num_probes=8, num_search_each_probe=10,␣
↪simulator=simulator(), score='EI',
                                                  interval=2, num_rand_basis=100)
```

3.5.4 Checking results

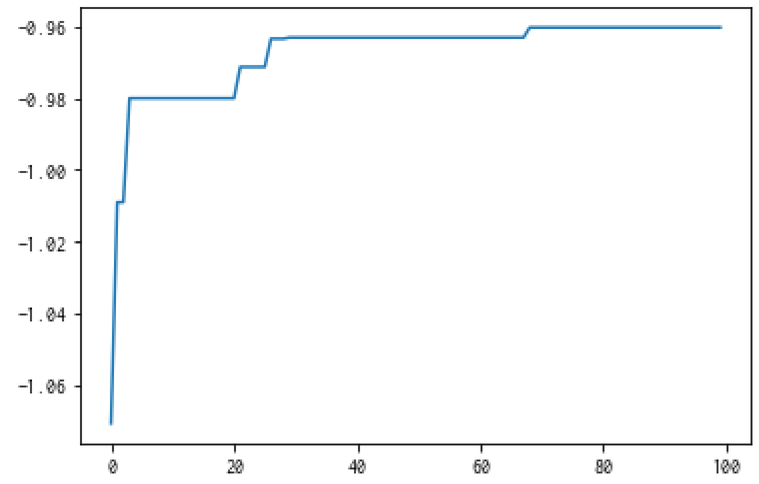
```
[6]: plt.plot(res.fx[0:res.total_num_search])
```

```
[6]: [<matplotlib.lines.Line2D at 0x7f9950ec84f0>]
```



```
[7]: best_fx, best_action = res.export_all_sequence_best_fx()
plt.plot(best_fx)
```

```
[7]: [<matplotlib.lines.Line2D at 0x7f9920ef7460>]
```

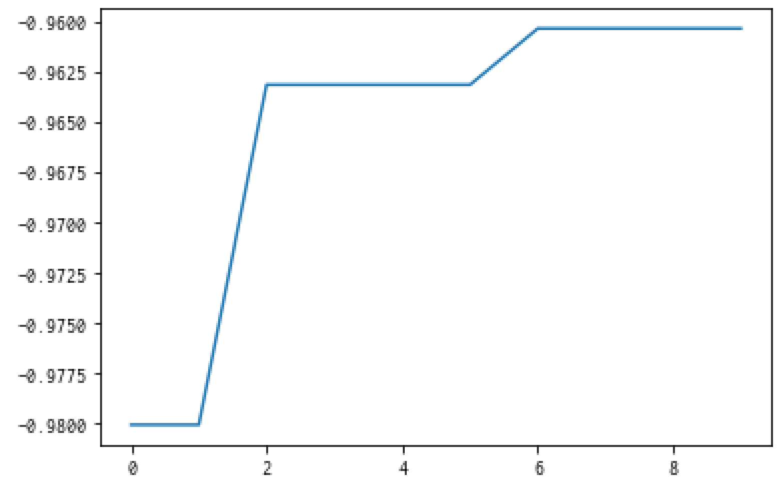


With `res.export_sequence_best_fx()`, you can get the best value obtained at each step and the history of the action.

The difference between `res.export_all_sequence_best_fx()` and `res.export_all_sequence_best_fx()` is that the information is not for each evaluation of the simulator, but for each search step. In this case, the total number of steps is 10, and the number of evaluations is 100.

```
[8]: best_fx, best_action = res.export_sequence_best_fx()
     plt.plot(best_fx)
```

```
[8]: [<matplotlib.lines.Line2D at 0x7f9930b43f10>]
```



3.6 Multi-objective optimization

When there are multiple objective functions (p) to be optimized, multi-objective optimization is used.
Note that in this tutorial, “solution” means the pair of objective functions $y = (y_1(x), y_2(x), \dots, y_p(x))$.
Define the solution size relation $\prec$ as follows.

$$y \prec y^{'} \iff \forall\, i \le p, y_i \le y^{'}_i \land \exists\, j \le p, y_j < y^{'}_j$$

A **Pareto solution** (in the maximization problem) is one in which no solution is larger than itself satisfying the above relations.
In other words, if you try to improve the value of any objective function, one of the other objective functions will deteriorate.
When there is a trade-off between objective functions, there are multiple Pareto solutions, and the numerical task is to find them efficiently.

PHYSBO implements a Bayesian optimization method to find Pareto solutions efficiently.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import physbo
%matplotlib inline
```

3.6.1 Test functions

In this tutorial, we will use VLMOP2, which is a benchmark function for multi-objective optimization.
The number of objective functions is two.

$$\begin{aligned} y_1(\vec{x}) &= 1 - \exp\left[-\sum_{i=1}^N \left(x_i - 1/\sqrt{N}\right)^2\right] \\ y_2(\vec{x}) &= 1 - \exp\left[-\sum_{i=1}^N \left(x_i + 1/\sqrt{N}\right)^2\right], \end{aligned}$$

where y_1 and y_2 have minimums at $x_1 = x_2 = \cdots x_N = 1/\sqrt{N}$ and $x_1 = x_2 = \cdots x_N = -1/\sqrt{N}$, respectively, both of which are 0. Also, the upper bound is 1.

Since PHYSBO solves a maximization problem, the objective function is again multiplied by -1.

- Refernce
  - Van Veldhuizen, David A. Multiobjective evolutionary algorithms: classifications, analyses, and new innovations. No. AFIT/DS/ENG/99-01. AIR FORCE INST OF TECH WRIGHT-PATTERSONAFB OH SCHOOL OF ENGINEERING, 1999.

[2]:
```
def vlmop2_minus(x):
    n = x.shape[1]
    y1 = 1 - np.exp(-1 * np.sum((x - 1/np.sqrt(n)) ** 2, axis = 1))
    y2 = 1 - np.exp(-1 * np.sum((x + 1/np.sqrt(n)) ** 2, axis = 1))
```
(continues on next page)

(continued from previous page)

```
    return np.c_[-y1, -y2]
```

3.6.2 Preparation of search candidate data

Let the input space $\vec{x}$ be two-dimensional, and generate a grid of candidate points on [-2, 2] × [-2, 2].

[3]:
```
import itertools
a = np.linspace(-2,2,101)
test_X = np.array(list(itertools.product(a, a)))
```

[4]:
```
test_X
```

[4]:
```
array([[-2.  , -2.  ],
       [-2.  , -1.96],
       [-2.  , -1.92],
       ...,
       [ 2.  ,  1.92],
       [ 2.  ,  1.96],
       [ 2.  ,  2.  ]])
```

[5]:
```
test_X.shape
```

[5]:
```
(10201, 2)
```

3.6.3 Definition of simulator

[6]:
```
class simulator(object):
    def __init__(self, X):
        self.t = vlmop2_minus(X)

    def __call__( self, action):
        return self.t[action]
```

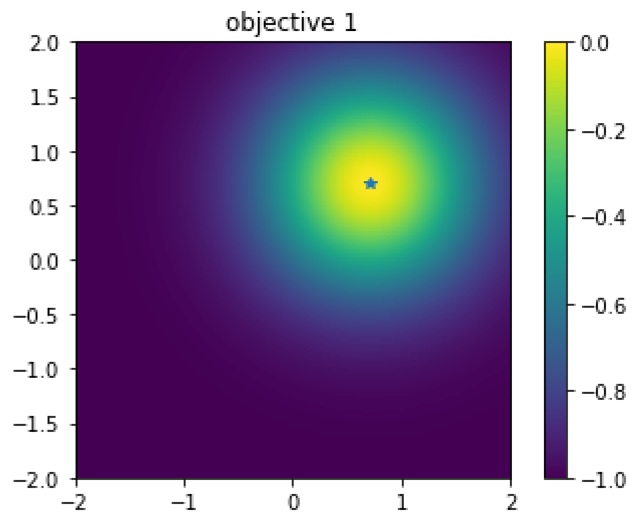
[7]:
```
simu = simulator(test_X)
```

3.6.4 Plotting the functions

Let's plot each of the two objective functions. The first objective function has a peak in the upper right corner, and the second objective function has a trade-off with a peak in the lower left corner (The star is the position of the peak.).

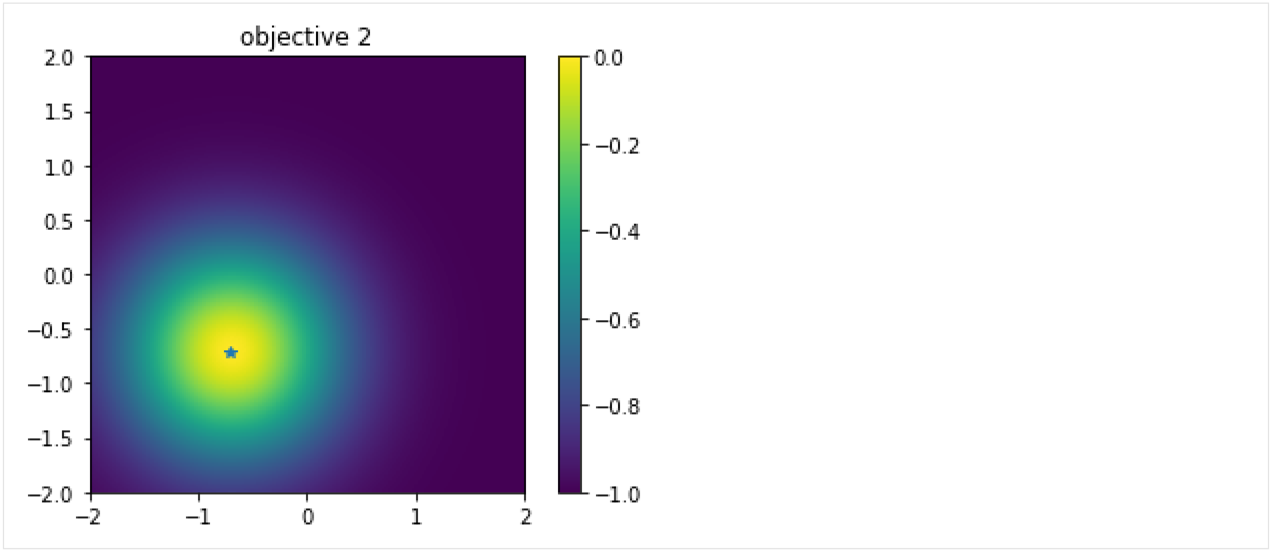
First objective function

[8]:
```
plt.figure()
plt.imshow(simu.t[:,0].reshape((101,101)), vmin=-1.0, vmax=0.0, origin="lower", extent=[-
↪2.0, 2.0, -2.0, 2.0])
plt.title("objective 1")
plt.colorbar()
plt.plot([1.0/np.sqrt(2.0)], [1.0/np.sqrt(2.0)], '*')
plt.show()
```



Second objective function

[9]:
```
# plot objective 2
plt.figure()
plt.imshow(simu.t[:,1].reshape((101,101)), vmin=-1.0, vmax=0.0, origin="lower", extent=[-
↪2.0, 2.0, -2.0, 2.0])
plt.title("objective 2")
plt.colorbar()
plt.plot([-1.0/np.sqrt(2.0)], [-1.0/np.sqrt(2.0)], '*')
plt.show()
```



3.6.5 Performing optimizations.

3.6.6 Setting policy

Use `physbo.search.discrete_multi.policy` for multi-objective optimization.
Specify the number of objective functions in `num_objectives`.

[10]:
```
policy = physbo.search.discrete_multi.policy(test_X=test_X, num_objectives=2)
policy.set_seed(0)
```

As with the usual usage of `physbo.search.discrete.policy` (with one objective function), optimization is done by calling the `random_search` or `bayes_search` methods. The basic API and usage are roughly the same as `discrete.policy`.

Random search

[ ]:
```
policy = physbo.search.discrete_multi.policy(test_X=test_X, num_objectives=2)
policy.set_seed(0)

res_random = policy.random_search(max_num_probes=50, simulator=simu)
```

The evaluation value of the objective function (the array) and the action ID at that time are displayed.
It also displays a message when the Pareto set is updated.

If you want to display the contents of the Pareto set when it is updated, specify `disp_pareto_set=True`.
Pareto set is sorted in ascending order of the first objective function value.

```
[ ]: policy = physbo.search.discrete_multi.policy(test_X=test_X, num_objectives=2)
     policy.set_seed(0)
     res_random = policy.random_search(max_num_probes=50, simulator=simu, disp_pareto_
     ↪set=True)
```

Checking results

History of evaluation values

```
[ ]: res_random.fx[0:res_random.num_runs]
```

Obtaining the Pareto solution

```
[14]: front, front_num = res_random.export_pareto_front()
      front, front_num
```

```
[14]: (array([[-0.95713719, -0.09067194],
              [-0.92633083, -0.29208351],
              [-0.63329589, -0.63329589],
              [-0.52191048, -0.72845916],
              [-0.26132949, -0.87913689],
              [-0.17190645, -0.91382463]]),
       array([40,  3, 19, 16, 29, 41]))
```

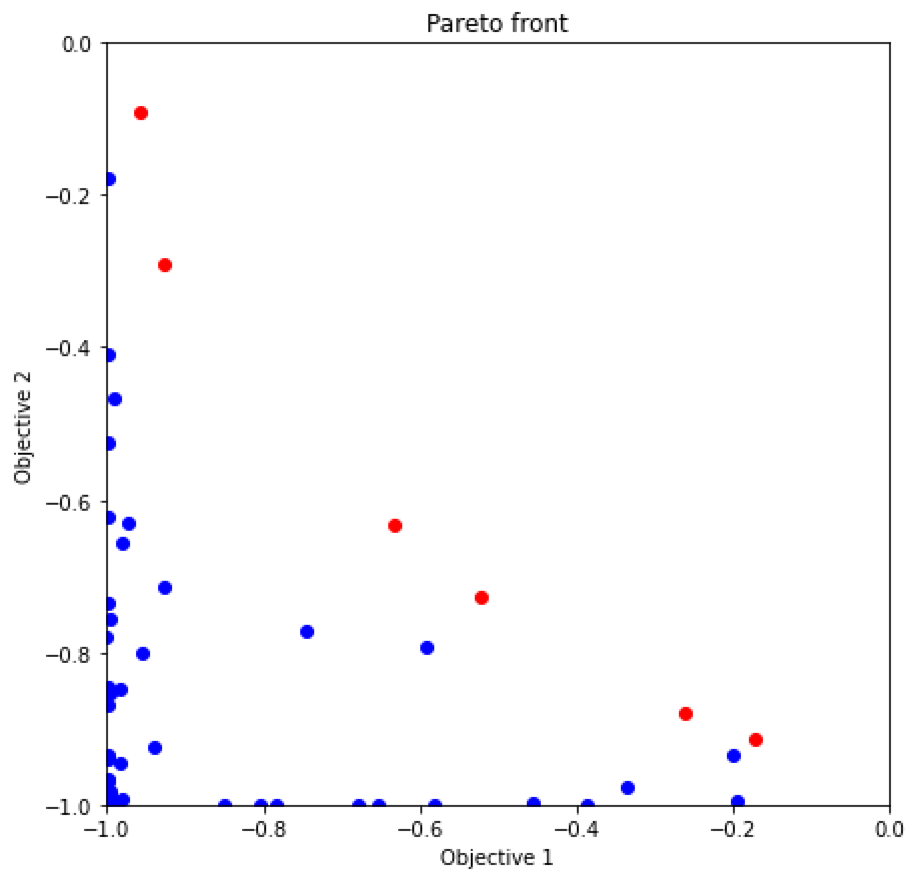
Plotting the solution (evaluated value)

Note again that the space to be plotted is $y = (y_1, y_2)$ and not $x = (x_1, x_2)$.

The red plot is the Pareto solution.

```
[15]: def plot_pareto_front(res):
          front, front_num = res.export_pareto_front()
          dominated = [i for i in range(res.num_runs) if i not in front_num]
          points = res.fx[dominated, :]

          plt.figure(figsize=(7, 7))
          plt.scatter(res.fx[dominated,0], res.fx[dominated,1], c = "blue")
          plt.scatter(front[:, 0], front[:, 1], c = "red")
          plt.title('Pareto front')
          plt.xlabel('Objective 1')
          plt.ylabel('Objective 2')
          plt.xlim([-1.0,0.0])
          plt.ylim([-1.0,0.0])
```

```
[16]: plot_pareto_front(res_random)
```



Calculate the volume of the dominated region

A solution that is not a Pareto solution, i.e., a solution y for which there exists a solution y' that is better than itself, is called a subordinate solution ($\exists y' y \prec y'$). The volume of the inferior solution region, which is the space occupied by inferior solutions in the solution space (a subspace of the solution space), is one of the indicators of the results of multi-objective optimization. The larger this value is, the more good Pareto solutions are obtained.`res_random.pareto.volume_in_dominance(ref_min, ref_max)` calculates the volume of the inferior solution region in the hyper-rectangle specified by `ref_min` and `ref_max`.

[17]:
```
res_random.pareto.volume_in_dominance([-1,-1],[0,0])
```

[17]:
```
0.2376881844865093
```

3.6.7 Bayesian optimization

For `bayes_search` in the multi-objective case, `score` can be selected from the following method

- HVPI (Hypervolume-based Probability of Improvement)
- EHVI (Expected Hyper-Volume Improvement)
- TS (Thompson Sampling)

The following 50 evaluations (10 random searches + 40 Bayesian optimizations) will be performed with different scores.

HVPI (Hypervolume-based Probability of Improvement)

The improvement probability of a non-dominated region in a multi-dimensional objective function space is obtained as a score.

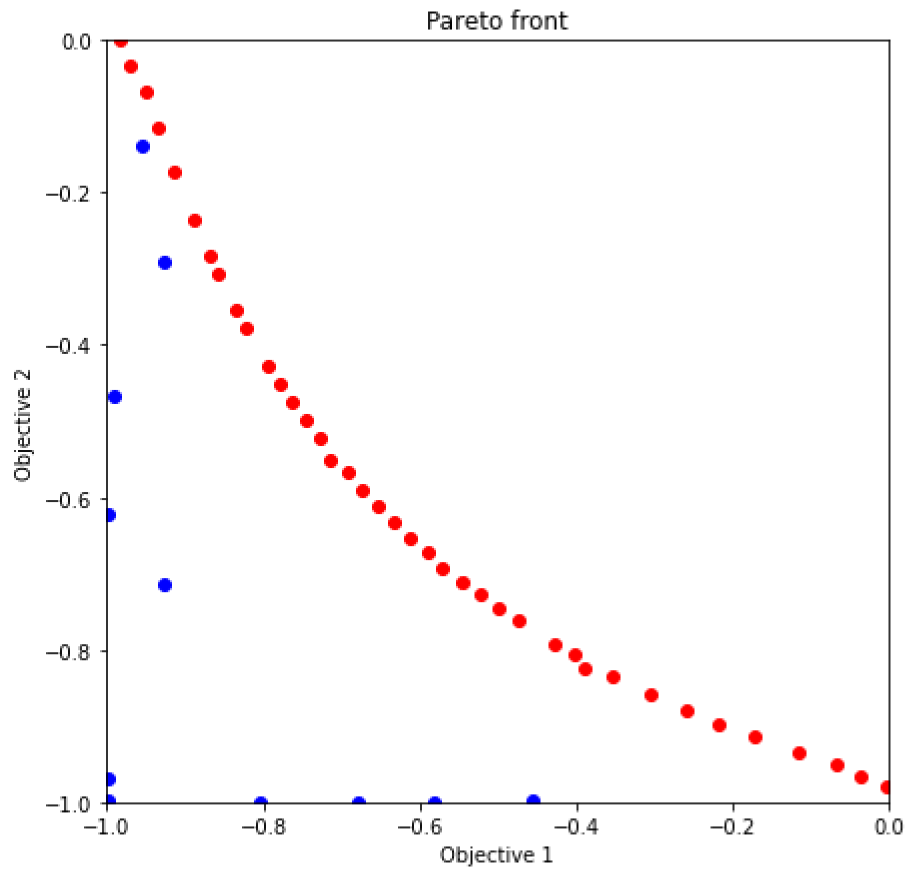
- Reference
  - Couckuyt, Ivo, Dirk Deschrijver, and Tom Dhaene. "Fast calculation of multiobjective probability of improvement and expected improvement criteria for Pareto optimization." Journal of Global Optimization 60.3 (2014): 575-594.

[ ]:
```
policy = physbo.search.discrete_multi.policy(test_X=test_X, num_objectives=2)
policy.set_seed(0)

policy.random_search(max_num_probes=10, simulator=simu)
res_HVPI = policy.bayes_search(max_num_probes=40, simulator=simu, score='HVPI',␣
↪interval=10)
```

Plotting the Pareto solution

We can see that more Pareto solutions are obtained compared to random sampling.

[19]:
```
plot_pareto_front(res_HVPI)
```



Volume of dominated region

```
[20]: res_HVPI.pareto.volume_in_dominance([-1,-1],[0,0])
```

```
[20]: 0.32877907991633726
```

EHVI (Expected Hyper-Volume Improvement)

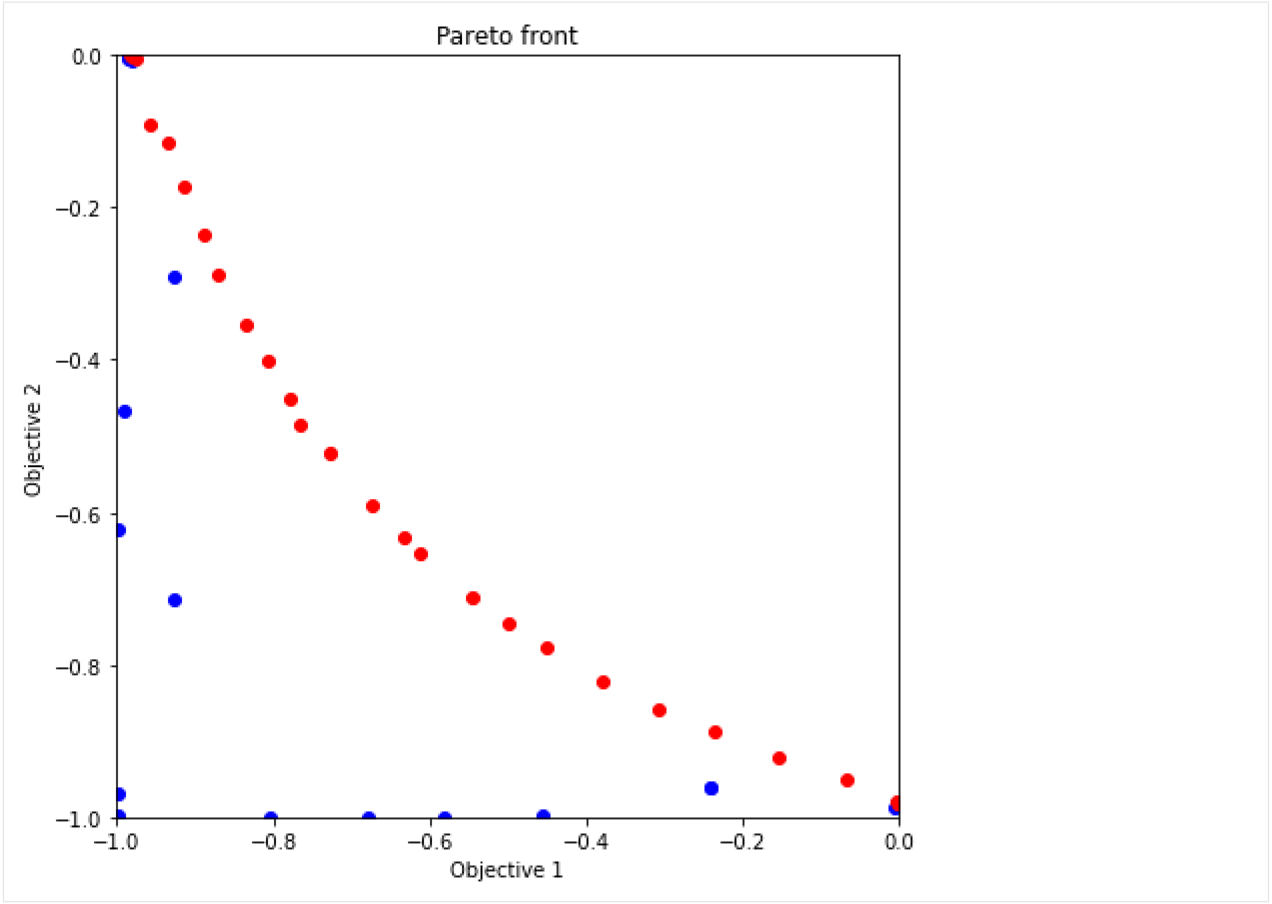
The expected improvement of the non-dominated region in the multi-dimensional objective function space is obtained as score.

- Reference
  - Couckuyt, Ivo, Dirk Deschrijver, and Tom Dhaene. "Fast calculation of multiobjective probability of improvement and expected improvement criteria for Pareto optimization." Journal of Global Optimization 60.3 (2014): 575-594.

```
[ ]: policy = physbo.search.discrete_multi.policy(test_X=test_X, num_objectives=2)
     policy.set_seed(0)

     policy.random_search(max_num_probes=10, simulator=simu)
     res_EHVI = policy.bayes_search(max_num_probes=40, simulator=simu, score='EHVI',␣
     ↪interval=10)
```

Plotting the Pareto solution

```
[22]: plot_pareto_front(res_EHVI)
```



Volume of dominated region

```
[23]: res_EHVI.pareto.volume_in_dominance([-1,-1],[0,0])
```

```
[23]: 0.3200467412741881
```

TS (Thompson Sampling)

In Thompson Sampling for the single objective case, at each candidate (test_X), sampling is performed from the posterior distribution of the objective function, and the candidate with the largest value is recommended as the next search point. In the multi-objective case, one candidate is randomly selected as the next search point from among the candidates with the maximum value based on the Pareto rule for the sampled values, i.e., the Pareto-optimal candidates.

- Reference
  - Yahyaa, Saba Q., and Bernard Manderick. “Thompson sampling for multi-objective multi-armed bandits problem.” Proc. Eur. Symp. Artif. Neural Netw., Comput. Intell. Mach. Learn.. 2015.

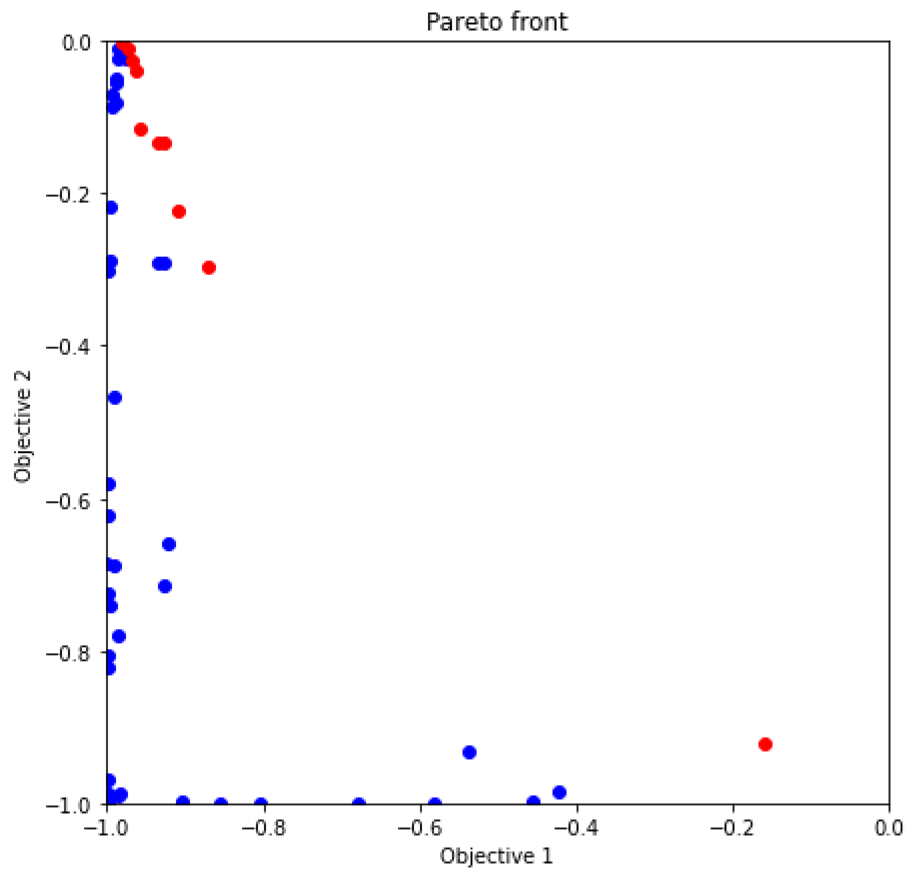
```
[ ]: policy = physbo.search.discrete_multi.policy(test_X=test_X, num_objectives=2)
policy.set_seed(0)

policy.random_search(max_num_probes=10, simulator=simu)
res_TS = policy.bayes_search(max_num_probes=40, simulator=simu, score='TS', interval=10,␣
↪num_rand_basis=5000)
```

(continues on next page)

(continued from previous page)

Plotting the Pareto solution

```
[25]: plot_pareto_front(res_TS)
```



Volume of dominated region

```
[26]: res_TS.pareto.volume_in_dominance([-1,-1],[0,0])
```

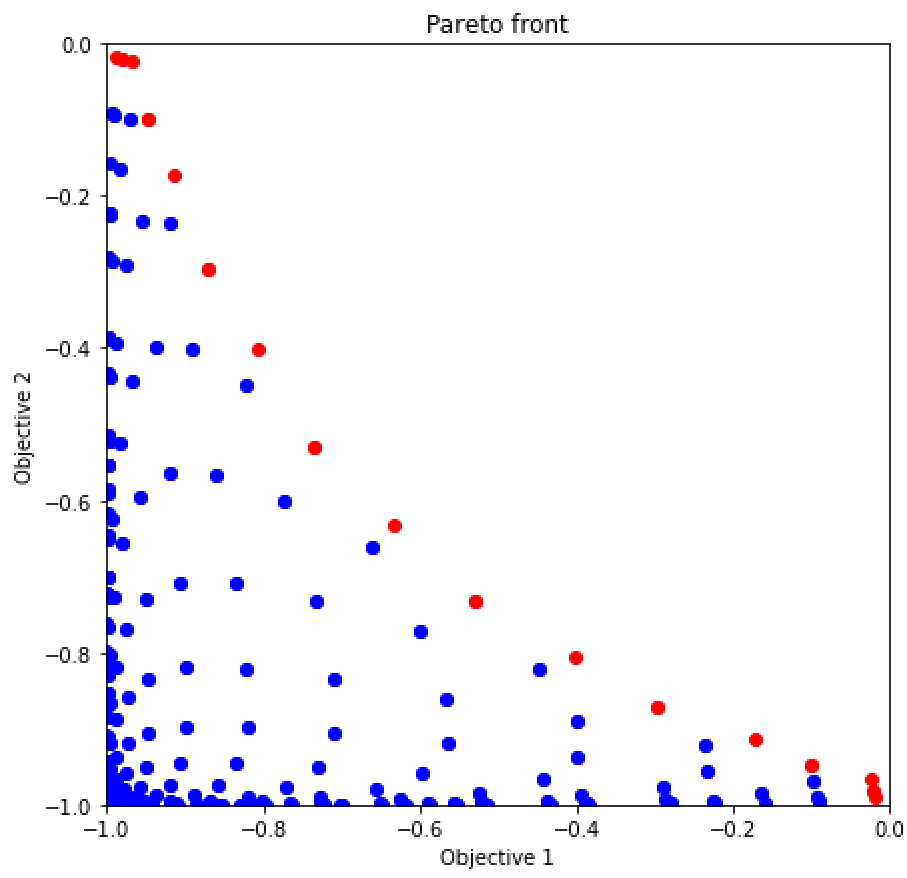
```
[26]: 0.16415446221006114
```

3.6.8 Appendix: Full search

In random_search, you can easily do a full search by passing the number of all data (N = test_X.shape[0]) to max_num_probes.
Since it takes time to evaluate all data, reduce the number of data in advance.

[ ]:
```
test_X_sparse = np.array(list(itertools.product(np.linspace(-2, 2, 21), repeat=2)))
simu_sparse = simulator(test_X_sparse)

policy = physbo.search.discrete_multi.policy(test_X=test_X_sparse, num_objectives=2)
policy.set_seed(0)

N = test_X_sparse.shape[0]
res_all = policy.random_search(max_num_probes=N, simulator=simu_sparse)
```

[28]:
```
plot_pareto_front(res_all)
```



[29]:
```
res_all.pareto.volume_in_dominance([-1,-1],[0,0])
```

[29]:
```
0.30051687493437484
```

CHAPTER

FOUR

ALGORITHM

This section describes an overview of Bayesian optimization. For technical details, please refer to this reference .

4.1 Bayesian optimization

Bayesian optimization is a method that can be used in complex simulations or real-world experimental tasks where the evaluation of the objective function (e.g., property values) is very costly. In other words, Bayesian optimization solves the problem of finding explanatory variables (material composition, structure, process and simulation parameters, etc.) that have a better objective function (material properties, etc.) with as few experiments and simulations as possible. In Bayesian optimization, we start from a situation where we have a list of candidates for the explanatory variables to be searched (represented by the vector $\mathbf{x}$). Then, from among the candidates, the one that is expected to improve the objective function y is selected by making good use of prediction by machine learning (using Gaussian process regression). We then evaluate the value of the objective function by performing experiments and simulations on the candidates. By repeating the process of selection by machine learning and evaluation by experimental simulation, optimization can be achieved in as few times as possible.

The details of the Bayesian optimization algorithm are described below.

- Step1: Initialization

Prepare the space to be explored in advance. In other words, list up the composition, structure, process, simulation parameters, etc. of the candidate materials as a vector $\mathbf{x}$. At this stage, the value of the objective function is not known. A few candidates are chosen as initial conditions and the value of the objective function y is estimated by experiment or simulation. This gives us the training data $D = \{\mathbf{x}_i, y_i\}_{(i=1,\cdots,N)}$ with the explanatory variables $\mathbf{x}$ and the objective function y.

- Step2: Selection of candidates

Using the training data, learn a Gaussian process. For Gaussian process, the mean of the predictions at arbitary $\mathbf{x}$ is $\mu_c(\mathbf{x})$ and the variance is $\sigma_c(\mathbf{x})$ are given as follows

$$\mu_c(\mathbf{x}) = \mathbf{k}(\mathbf{x})^T(K+\sigma^2 I)^{-1}\mathbf{y},$$
$$\sigma_c(\mathbf{x}) = k(\mathbf{x},\mathbf{x}) + \sigma^2 - \mathbf{k}(\mathbf{x})^T(K+\sigma^2 I)^{-1}\mathbf{k}(\mathbf{x}),$$

where $k(\mathbf{x},\mathbf{x}')$ is a function called as a kernel, and it represents the similarity of two vectors. In general, the following Gaussian kernel is used:

$$k(\mathbf{x},\mathbf{x}') = \exp\left[-\frac{1}{2\eta^2}||\mathbf{x}-\mathbf{x}'||^2\right].$$

Using this kernel function, $\mathbf{k}(\mathbf{x})$ and K are computed as follows

$$\mathbf{k}(\mathbf{x}) = (k(\mathbf{x}_1,\mathbf{x}), k(\mathbf{x}_2,\mathbf{x}), \cdots, k(\mathbf{x}_N,\mathbf{x}))^\top$$

$$K = \begin{pmatrix} k(\mathbf{x}_1, \mathbf{x}_1) & k(\mathbf{x}_1, \mathbf{x}_2) & \ldots & k(\mathbf{x}_1, \mathbf{x}_N) \\ k(\mathbf{x}_2, \mathbf{x}_1) & k(\mathbf{x}_2, \mathbf{x}_2) & \ldots & k(\mathbf{x}_2, \mathbf{x}_N) \\ \vdots & \vdots & \ddots & \vdots \\ k(\mathbf{x}_N, \mathbf{x}_1) & k(\mathbf{x}_N, \mathbf{x}_2) & \ldots & k(\mathbf{x}_N, \mathbf{x}_N) \end{pmatrix}$$

For all candidates that have not yet been tested or simulated, the prediction $\mu_c(\mathbf{x})$ and the variance associated with the uncertainty of the prediction $\sigma_c(\mathbf{x})$ are estimated. Using this, the acquisition function is calculated. Then, the candidate $\mathbf{x}^*$ is selected that maximizes the acquisition function from among the candidates for which we do not yet know the value of the objective function. In this case, σ and η are called hyperparameters, and PHYSBO will automatically set the best value.

As an acquisition function, for example, Maximum Probability of Improvement (PI) and Maximum Expected Improvement (EI) are useful. The score of PI is defined as follows.

$$\mathrm{PI}(\mathbf{x}) = \Phi(z(\mathbf{x})), \quad z(\mathbf{x}) = \frac{\mu_c(\mathbf{x}) - y_{\max}}{\sigma_c(\mathbf{x})},$$

where $\Phi(\cdot)$ is the cumulative distribution function. The PI score represents the probability of exceeding the maximum $y_{\max}$ of the currently obtained y. In addition, the EI score is the expected value of the difference between the predicted value and the current maximum $y_{\max}$ and is given by

$$\mathrm{EI}(\mathbf{x}) = [\mu_c(\mathbf{x}) - y_{\max}]\Phi(z(\mathbf{x})) + \sigma_c(\mathbf{x})\phi(z(\mathbf{x})), \quad z(\mathbf{x}) = \frac{\mu_c(\mathbf{x}) - y_{\max}}{\sigma_c(\mathbf{x})},$$

where $\phi(\cdot)$ is a probability density function.

- Step3: Experiment (Simulation)

Perform an experiment or simulation on the candidate $\mathbf{x}^*$ with the largest acquisition function selected in step 2, and estimate the objective function value y. This will add one more piece of training data. Repeat steps 2 and 3 to search for candidates with good scores.

4.2 Accelerating Bayesian Optimization with PHYSBO

In PHYSBO, random feature map, Thompson sampling, and Cholesky decomposition are used to accelerate the calculation of Bayesian optimization. First, the random feature map is introduced. By introducing the random feature map $\phi(\mathbf{x})$, we can approximate the Gaussian kernel $k(\mathbf{x}, \mathbf{x}')$ as follows.

$$k(\mathbf{x}, \mathbf{x}') = \exp\left[-\frac{1}{2\eta^2}\|\mathbf{x} - \mathbf{x}'\|\right]^2 \simeq \phi(\mathbf{x})^\top \phi(\mathbf{x}')$$
$$\phi(\mathbf{x}) = (z_{\omega_1, b_1}(\mathbf{x}/\eta), ..., z_{\omega_l, b_l}(\mathbf{x}/\eta))^\top,$$

where $z_{\omega,b}(\mathbf{x}) = \sqrt{2}\cos(\boldsymbol{\omega}^\top \mathbf{x} + b)$. Then, $\boldsymbol{\omega}$ is generated from $p(\boldsymbol{\omega}) = (2\pi)^{-d/2}\exp(-\|\boldsymbol{\omega}\|^2/2)$ and b is chosen uniformly from $[0, 2\pi]$ is chosen uniformly from $[0, 2\pi]$. This approximation is strictly valid in the limit of $l \to \infty$, where the value of l is the dimension of the random feature map.

> Φ can be represented as a l row n column matrix with $\phi(\mathbf{x}_i)$ in each column by $\mathbf{x}$ vector of training data as follows:
>
> $$\Phi = (\phi(\mathbf{x}_1), ..., \phi(\mathbf{x}_n)).$$

It is seen that the following relation is satisfied:

$$\mathbf{k}(\mathbf{x}) = \Phi^\top \phi(\mathbf{x})$$
$$K = \Phi^\top \Phi.$$

Next, a method that uses Thompson sampling to make the computation time for candidate prediction $O(l)$ is introduced. Note that using EI or PI will result in $O(l^2)$ because of the need to evaluate the variance. In order to perform Thompson sampling, the Bayesian linear model defined below is used.

$$y = \mathbf{w}^\top \phi(\mathbf{x}),$$

where $\phi(\mathbf{x})$ is random feature map described above and $\mathbf{w}$ is a coefficient vector. In a Gaussian process, when the training data D is given, this $\mathbf{w}$ is determined to follow the following Gaussian distribution.

$$\begin{aligned} p(\mathbf{w}|D) &= \mathcal{N}(\boldsymbol{\mu}, \Sigma) \\ \boldsymbol{\mu} &= (\Phi\Phi^\top + \sigma^2 I)^{-1}\Phi\mathbf{y} \\ \Sigma &= \sigma^2(\Phi\Phi^\top + \sigma^2 I)^{-1} \end{aligned}$$

In Thompson sampling, one coefficient vector is sampled according to this posterior probability distribution and set to $\mathbf{w}^*$, which represents the acquisition function as follows

$$\mathrm{TS}(\mathbf{x}) = {\mathbf{w}^*}^\top \phi(\mathbf{x}).$$

The $\mathbf{x}^*$ that maximizes $\mathrm{TS}(\mathbf{x})$ will be selected as the next candidate. In this case, $\phi(\mathbf{x})$ is an l dimensional vector, so the acquisition function can be computed with $O(l)$.

Next, the manner for accelerating the sampling of $\mathbf{w}$ is introduced. The matrix A is defined as follows.

$$A = \frac{1}{\sigma^2}\Phi\Phi^\top + I$$

Then the posterior probability distribution is given as

$$p(\mathbf{w}|D) = \mathcal{N}\left(\frac{1}{\sigma^2}A^{-1}\Phi\mathbf{y}, A^{-1}\right).$$

Therefore, in order to sample $\mathbf{w}$, we need to calculate A^{-1}. Now consider the case of the newly added $(\mathbf{x}', y')$ in the Bayesian optimization iteration. With the addition of this data, the matrix A is updated as

$$A' = A + \frac{1}{\sigma^2}\phi(\mathbf{x}')\phi(\mathbf{x}')^\top.$$

This update can be done using the Cholesky decomposition ($A = L^\top L$), which reduces the time it takes to compute A^{-1} to $O(l^2)$. If we compute A^{-1} at every step, the numerical cost becomes $O(l^3)$. The $\mathbf{w}$ is obtained by

$$\mathbf{w}^* = \boldsymbol{\mu} + \mathbf{w}_0,$$

where $\mathbf{w}_0$ is sampled from $\mathcal{N}(0, A^{-1})$ and $\boldsymbol{\mu}$ is calculated by

$$L^\top L\boldsymbol{\mu} = \frac{1}{\sigma^2}\Phi\mathbf{y}.$$

By using these techniques, a computation time becomes almost linear in the number of training data.

CHAPTER

FIVE

ACKNOWLEDGEMENT

We would like to thank the support from “Project for advancement of software usability in materials science” by The Institute for Solid State Physics, The University of Tokyo, for development of PHYSBO.

CONTACT

- Bug Reports

  Please report all problems and bugs on the github Issues page.

  To resolve bugs early, follow these guidelines when reporting:

  1. Please specify the version of PHYSBO you are using.
  2. If there are problems for installation, please inform us about your operating system and the compiler.
  3. If a problem occurs during execution, enter the input file used for execution and its output.

  Thank you for your cooperation.

- Others

  If you have any questions about your research that are difficult to consult at Issues on GitHub, please send an e-mail to the following address:

  E-mail: `physbo-dev__at__issp.u-tokyo.ac.jp` (replace _at_ by @)