\Orchestrating a brighter world NEC

NEC Vector Annealing Parameter Estimator User's Guide

6th Edition

Precautions When Exporting

This product (including the software) may correspond to a regulated good (or service) as defined by the Foreign Exchange and Foreign Trade Act.

In that case, an export license from the Japanese government is required when exporting the product outside of Japan.

If related documentation, etc. is required for the export license application procedure, please consult with the dealer from which the product was purchased or a nearby NEC sales office.

Preface

- This document explains how to use Parameter Estimator with NEC Vector Annealing.

- Trademarks and copyrights
  - Other listed company names and product names are registered trademarks or trademarks of their respective owners.

Terminology Definitions

Terminology	Description
constraint mode	In this mode, the search prioritizes satisfying constraints. Specify vector_mode=VectorAnnealing.VECTOR_MODE_CONSTRAINT.
non-constraint mode	In this mode, the search prioritizes finding solutions that minimize energy. Specify vector_mode=VectorAnnealing.VECTOR_MODE_SPEED.

Table of contents

1. Overview of the Parameter Estimator
2. Installation Instructions
3. Usage
4. Specifications
5. Notes for Use

1. Overview of the Parameter Estimator

Overview

- What is the Parameter Estimator?
  - When using NEC Vector Annealing (VA), the solution performance varies greatly depending on the value of the input parameter of inverse temperature (beta). Also, if the Hamiltonian consists of multiple terms, the solution performance varies greatly depending on the value of the weight coefficient multiplied by each term. Tuning of these parameters is necessary to obtain the best performance.

  - The Parameter Estimator is a tool that estimates the appropriate values of those parameters by analyzing equations created using PyQUBO*.

* : PyQUBO
A domain specific language (DSL) that converts a formulated formula to the QUBO format when solving a combinatorial optimization problem with an annealing machine.
It can be used in the same way as other Python libraries.
https://pyqubo.readthedocs.io/en/latest/index.html
https://github.com/recruit-communications/pyqubo

Output value

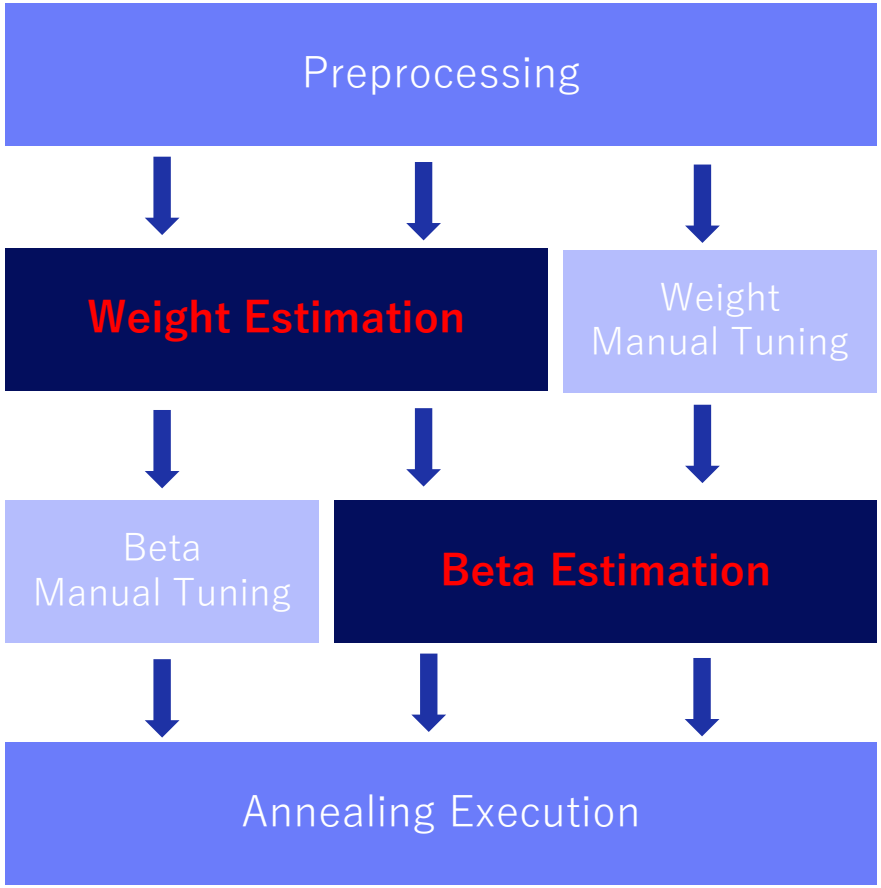
- Values that can be output by the Parameter Estimator

Output item	Description
beta (beta_list or beta_range)	Automatically estimated inverse temperature data. Can be output in the form of a list of all estimated inverse temperatures (beta_list) or in the form of [first, last, total] (beta_range) .
weight (feed_dict)	Dictionary of weight coefficient values to multiply the Hamiltonian constraint terms. The output can be in the dictionary format of {term name : weight, ...} .

Procedure for use

- Follow this procedure.
  1. Preprocessing (importing, creating expressions)
  2. Weight Estimation
  3. Beta Estimation
  4. Annealing Execution

- By using the Parameter Estimator, it is possible to estimate weights and betas from the above.
- The Parameter Estimator does not estimate the number of sweeps for solving. Please adjust it manually.
- It is possible to estimate both weights and betas, or to estimate only one and adjust the other manually.
- Please estimate weights first, then estimate betas.



Functionality of the Parameter Estimator – API List

- The parameter estimator provides the following APIs:

- API for estimating weights

API name	Description
get_feed()	This API is used in non-constraint mode. It estimates weights (feed_dict). Note that weight estimation cannot be performed in constraint mode.

- API for estimating betas

API name	Description
get_beta()	This API is used in non-constraint mode. It estimates the beta in list format ([beta0, beta1, ..., betan]).
get_beta_range()	This API is used in non-constraint mode. It estimates the beta in range format([start, end, step]).
get_constraint_beta()	This API is used in constraint mode. It estimates the beta in list format ([beta0, beta1, ..., betan]).
get_constraint_beta_range()	This API is used in constraint mode. It estimates the beta in range format([start, end, step]).

2. Installation Instructions

Advance Preparation

What to prepare

NEC Vector Annealing (VA) execution environment

- Environment where NEC Vector Annealing is installed.

parameter_estimator-3.0.0

- parameter_estimator-3.0.0-cp311-cp311-linux_x86_64.whl provided separately
- Please select the wheel package according to the OS of your runtime environment.
  - If the OS is RHEL 8.8/8.10 or Rocky Linux 8.8/8.10, choose the wheel package in the rhel8 folder.
  - If the OS is RHEL 9.2/9.4 or Rocky Linux 9.2/9.4, choose the wheel package in the rhel9 folder.

Python3.11, PyQUBO (version >= 1.4.0)

- Use versions of PyQUBO 1.4.0 or later.
- **The recommended version of PyQUBO is 1.4.0. Only Python 3.11 is supported.**

Parameter Estimator Installation

Installation procedure

Install with the pip command

- As shown below, place the wheel package in any directory (DIR) and install it with the pip command.

```
$ pip3.11 install DIR/parameter_estimator-3.0.0-cp311-cp311-linux_x86_64.whl --user
```

  - When installing in the user's personal environment, please specify the "--user" option.

- You can verify the installed version of Parameter Estimator with the pip command.

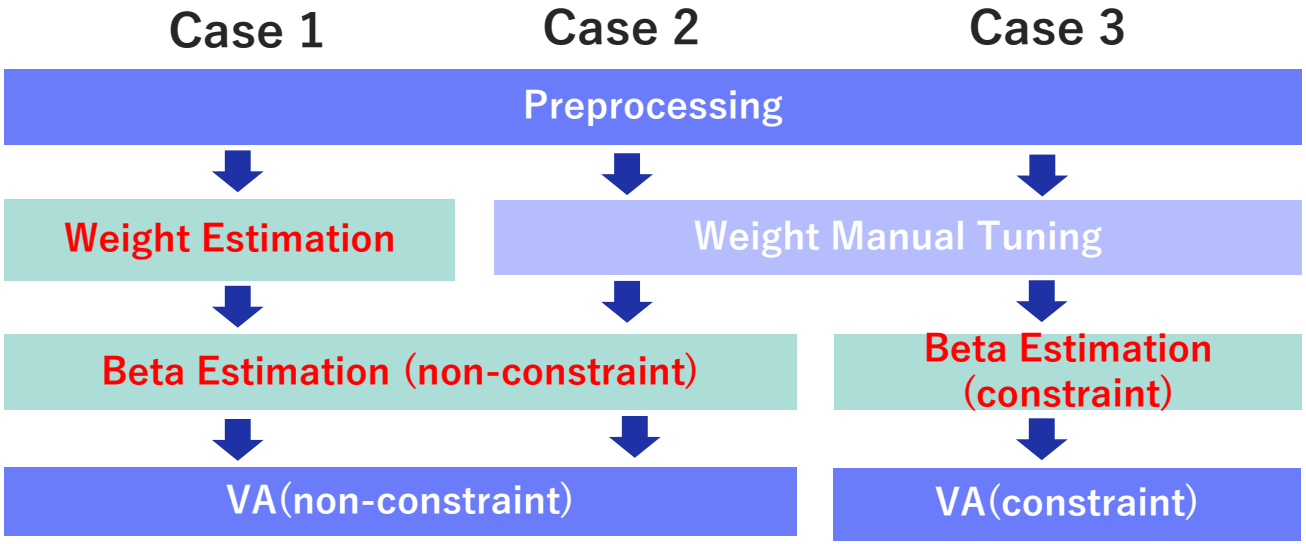
```
$ pip3.11 show parameter_estimator
Name: parameter-estimator
Version: 3.0.0 Check version
Summary: Parameter Estimator for NEC Vector Annealing
.
.
```

3. Usage

Procedure for use

- We assume usage in the following three cases. Examples of usage for each case are provided on the following pages.

Case	mode	Beta Estimation	Weight Estimation	Notes
Case 1	non-constraint	Do	Do	high_order is not supported.
Case 2	non-constraint	Do	Do not	
Case 3	constraint	Do	Do not	Estimation of weights cannot be done in constraint mode.



Case 1

- This is a case where weights and betas are estimated in non-constraint mode. Please follow the steps below for usage.
  - In this case, it is not possible to estimate a model that includes higher-order terms.

1. Importing the module
2. Creating the model
3. Creating the Estimator instance
4. Estimating weights
5. Estimating betas
6. Executing annealing

Case 1- Preprocessing

- Module import
  - Import parameter_estimator, pyqubo, and VectorAnnealing
    - Import the installed parameter_estimator.

```
import VectorAnnealing
from parameter_estimator  import  parameter_estimator
from pyqubo import Array, Placeholder, Constraint
```

Only vartype='BINARY' in Array.create is supported.
Others such as UnaryEncInteger and LogEncInteger are not supported.

Case 1 - Creating the Model

- Define the spin and set the objective function and constraints.
  - Define each constraint individually using Constraint().
  - Notes:
    - Labeling multiple constraints at once will not function correctly.
    - MIN(), MAX(), MINMAX(), and FREE() are reserved words and cannot be used as label names.

```
n = 5
x = Array.create('x', shape=n, vartype='BINARY')

# Define the objective function Hw
Hw = sum(sum(x[i]*x[j] for i in range(n)) for j in range(n))

# Define the objective function Hc
Hc = Constraint((sum(x[i] for i in range(n)) - 1) ** 2, label="onehot")

param_hw = Placeholder('hw')
param_hc = Placeholder("hc")
H = param_hw * Hw + param_hc * Hc
```

Case 1 - Creating the Model

- Setting Flip Options
  - Set each flip option according to the VA user's guide.
  - For example, when using onehot and fixed, set the flip options in constflip as shown below.

```
onehot = [[f'x[{i}]' for i in range(n)]]
fixed = [['x[0]', 0]]

constflip = {"onehot":onehot, "fixed":fixed}
```

(Reference) List of flip options supported by Paramter_Estimator

- If there are spins included only in the flip options, you need to add those spins to the spin_list.

Case 1 - Creating the Model

- Creating the model and qubo
  - Specify the temporary weights in feed_dict and create the model and qubo from the Hamiltonian.
    - In feed_dict, specify a dictionary with names of the weights set in Placeholder() as the key and positive real numbers as the value.
  - The weights of the objective function are not estimated. They will be fixed at the specified values. (If multiple objective functions are defined, please manually adjust the ratio of each objective function).

```
model = H.compile()
feed_dict = {"hw":1, "hc":1}
qubo, offset = model.to_qubo(feed_dict=feed_dict)
```

Case 1 - Creating the Model

- Creating the spin_list
  - Create the list of spins included in the model.
  - The list of spins used in the model created by PyQUBO is defined in model.variables. Please copy it.

```
spin_list = model.variables
```

  - If there are spins included only in the flip options, please add them manually.

```
y = Array.create('y', shape=3, vartype='BINARY')
orone = ['x[0]', 'y[0]', 'y[1]', 'y[2]']
constflip["orone"] = [orone]

# Add y[0], y[1], and y[2] specified in the orone flip option to spin_list
spin_list = list(set(spin_list) | set(orone))
```

  - Including unnecessary spins may result in incorrect estimations.

Case 1 - Creating an Instance

- Creating an instance of the Estimator class
  - Create an instance as shown below.

```
#Creating an instance
est = parameter_estimator.Estimator()
```

  - Notes: If the Hamiltonian is changed, you will need to recreate the instance. If you do not recreate the instance, there is a possibility of incorrect estimation.

Case 1 – Estimating the Weight

- Automatic Estimation of Weights with the Parameter Estimator
  - Use the get_feed() function to estimate weights (feed_dict).
  - Specify the same feed_dict as the provisional feed_dict specified in model.to_qubo() and the flip options defined in constflip as shown below.
  - Setting cal_all_feed to True will perform the estimation considering the objective function.

```
#Estimating the feed
feed_dict = est.get_feed(Hc, feed_dict, spin_list, qubo, **constflip, cal_all_feed=True)

#recalculate qubo
qubo, offset = model.to_qubo(feed_dict=feed_dict)
```

- After estimating the feed_dict, recalculate and update the qubo using that feed_dict (a qubo will be created with the estimated feed_dict as the coefficients).

Case 1 – Estimating the Beta

- Automatic Estimation of Betas with the Parameter Estimator
  - Use the get_beta() function or the get_beta_range() function to automatically estimate betas.

    - When estimating beta_range:

```
beta_range = est.get_beta_range(Hc, feed_dict, spin_list, qubo, num_sweeps=1000, **constflip)
```

    - When estimating beta_list:

```
beta_list = est.get_beta(Hc, feed_dict, spin_list, qubo, num_sweeps=1000, **constflip)
```

Case 1 – Execution of VA

- Use the recalculated qubo to compute the va_model, and input the va_model and the beta into va_sampler to perform the solution.
  - Make sure not to confuse beta_list with beta_range.
    - When estimating beta_list:

```
feed_dict = est.get_feed(Hc, feed_dict, spin_list, qubo, **constflip, cal_all_feed=True)
qubo, offset = model.to_qubo(feed_dict=feed_dict)
beta_list = est.get_beta(Hc, feed_dict, spin_list, qubo, num_sweeps=1000, **constflip)

va_model = VectorAnnealing.model(qubo, offset, **constflip)
va       = VectorAnnealing.sampler()
results = va.sample(va_model, num_sweeps=1000,
beta_list=beta_list,vector_mode=VectorAnnealing.VECTOR_MODE_SPEED)
```

    - When estimating beta_range:

```
feed_dict = est.get_feed(Hc, feed_dict, spin_list, qubo, **constflip, cal_all_feed=True)
qubo, offset = model.to_qubo(feed_dict=feed_dict)
beta_range = est.get_beta_range(Hc, feed_dict, spin_list, qubo, num_sweeps=1000, **constflip)

va_model = VectorAnnealing.model(qubo, offset, **constflip)
va       = VectorAnnealing.sampler()
results = va.sample(va_model, num_sweeps=1000, beta_range=beta_range,
vector_mode=VectorAnnealing.VECTOR_MODE_SPEED)
```

Case 1 – Sample Code

```
import VectorAnnealing
from parameter_estimator import parameter_estimator
from pyqubo import Array,Placeholder,Constraint


n= 5
x = Array.create('x', shape=n, vartype='BINARY')
Hw = sum(sum(x[i]*x[j] for i in range(n)) for j in range(n))
Hc = Constraint((sum(x[i] for i in range(1,n)) - 1) ** 2, label="onehot")
param_hw = Placeholder('hw')
param_hc = Placeholder("hc")
H = param_hw * Hw + param_hc * Hc

# Flip Option
onehot = [[f'x[{i}]' for i in range(1,n)]]
fixed = [['x[0]', 0]]
constflip = {"onehot":onehot, "fixed":fixed}

# create model
model = H.compile()
feed_dict = {"hw":1, "hc":1}
qubo, offset = model.to_qubo(feed_dict=feed_dict)
spin_list = model.variables

# paramter_estimator
est = parameter_estimator.Estimator()
feed_dict = est.get_feed(Hc, feed_dict, spin_list, qubo, **constflip, cal_all_feed=True)
qubo, offset = model.to_qubo(feed_dict=feed_dict)
beta_range = est.get_beta_range(Hc, feed_dict,spin_list, qubo, num_sweeps=1000, **constflip)

#VA
sa_model = VectorAnnealing.model(qubo, offset, **constflip)
sampler = VectorAnnealing.sampler()
results = sampler.sample(sa_model, num_sweeps=1000, beta_range=beta_range, vector_mode=VectorAnnealing.VECTOR_MODE_SPEED)
```

Case 2

- This is a case where betas are estimated in non-constraint mode. Please follow the steps below for usage.
  - In this case, it is possible to estimate a model that includes higher-order terms.

1. Importing the module
2. Creating the model
3. Creating the Estimator instance
4. Estimating betas
5. Executing annealing

Case 2- Preprocessing

- Module import
  - Import parameter_estimator, pyqubo, and VectorAnnealing
    - Import the installed parameter_estimator.

```
import VectorAnnealing
from parameter_estimator  import  parameter_estimator
from pyqubo import Array, Placeholder, Constraint
```

Only vartype='BINARY' in Array.create is supported.
Others such as UnaryEncInteger and LogEncInteger are not supported.

Case 2 - Creating the Model

- Define the spin and set the objective function and constraints.
  - Define each constraint individually using Constraint().
  - Notes:
    - Labeling multiple constraints at once will not function correctly.
    - MIN(), MAX(), MINMAX(), and FREE() are reserved words and cannot be used as label names.

```
n = 5
x = Array.create('x', shape=n, vartype='BINARY')

# Define the objective function Hw
Hw = sum(sum(x[i]*x[j] for i in range(n)) for j in range(n))

# Define the objective function Hc
Hc = Constraint((sum(x[i] for i in range(n)) - 1) ** 2, label="onehot")

param_hw = Placeholder('hw')
param_hc = Placeholder("hc")
H = param_hw * Hw + param_hc * Hc
```

Case 2 - Creating the Model

- Setting Flip Options
  - Set each flip option according to the VA user's guide.
  - For example, when using onehot and fixed, set the flip options in constflip as shown below.

```
onehot = [[f'x[{i}]' for i in range(n)]]
fixed = [['x[0]', 0]]

constflip = {"onehot":onehot, "fixed":fixed}
```

(Reference) List of flip options supported by Paramter_Estimator

- If there are spins included only in the flip options, you need to add those spins to the spin_list.

Case 2 - Creating the Model

- Creating the model and qubo
  - Create the model and qubo from the created Hamiltonian.
  - Prepare feed_dict separately.

```
model = H.compile()
feed_dict = {"hw":2, "hc":1}
qubo, offset = model.to_qubo(feed_dict=feed_dict)
```

- Creating higher-order terms
  - When estimating only beta, it is also possible to specify higher-order terms. (Optional)

```
high_order = {('z[0]', 'z[1]', 'z[2]'):1, ('z[0]', 'z[1]', 'x[0]'):2}
```

  - If there are spins included only in the higher-order terms, you will need to add them to the spin_list later.

Case 2 - Creating the Model

- Creating the spin_list
  - Create the list of spins included in the model.
  - The list of spins used in the model created by PyQUBO is defined in model.variables. Please copy it.

```
spin_list = model.variables
```

  - If there are spins included only in the flip options, please add them manually.

```
y = Array.create('y', shape=3, vartype='BINARY')
high_order = {('z[0]', 'z[1]', 'z[2]'):1, ('z[0]', 'z[1]', 'x[0]'):2}
orone = ['x[0]', 'y[0]', 'y[1]', 'y[2]']
constflip["orone"] = [orone]

# y[0], y[1], y[2], z[0], z[1], z[2]
spin_list = list(set(spin_list) | set(orone) | {'z[0]', 'z[1]', 'z[2]'})
```

  - Including unnecessary spins may result in incorrect estimations.

Case 2 - Creating an Instance

- Creating an instance of the Estimator class
  - Create an instance as shown below.

```
#Creating an instance
est = parameter_estimator.Estimator()
```

  - Notes: If the Hamiltonian is changed, you will need to recreate the instance. If you do not recreate the instance, there is a possibility of incorrect estimation.

Case 2 – Estimating the Beta

- Automatic Estimation of Betas with the Parameter Estimator
  - Use the get_beta() function or the get_beta_range() function to automatically estimate betas.

    - When estimating beta_range:

```
beta_range = est.get_beta_range(Hc, feed_dict, spin_list, qubo, num_sweeps=1000,
high_order=high_order, **constflip)
```

    - When estimating beta_list:

```
beta_list = est.get_beta(Hc, feed_dict, spin_list, qubo, num_sweeps=1000,
high_order=high_order,**constflip)
```

Case 2 – Execution of VA

- Use the recalculated qubo to compute the va_model, and input the va_model and the beta into va_sampler to perform the solution.
  - Make sure not to confuse beta_list with beta_range.
    - When estimating beta_list:

```
beta_list = est.get_beta(Hc, feed_dict, spin_list, qubo, num_sweeps=1000,
                         high_order=high_order, **constflip)

va_model = VectorAnnealing.model(qubo, offset, high_order=high_order, **constflip)
va       = VectorAnnealing.sampler()
results = va.sample(va_model, beta_list=beta_list, num_sweeps=1000,
                    vector_mode=VectorAnnealing.VECTOR_MODE_SPEED)
```

    - When estimating beta_range:

```
beta_range = est.get_beta_range(Hc, feed_dict, spin_list, qubo,
                                num_sweeps=1000,high_order=high_order,**constflip)

va_model = VectorAnnealing.model(qubo, offset, high_order=high_order, **constflip)
va       = VectorAnnealing.sampler()
results = va.sample(va_model, beta_range=beta_range, num_sweeps=1000,
                    vector_mode=VectorAnnealing.VECTOR_MODE_SPEED)
```

Case 2 - Sample Code

```
import VectorAnnealing
from parameter_estimator import parameter_estimator
from pyqubo import Array,Placeholder,Constraint


n= 5
x = Array.create('x', shape=n, vartype='BINARY')
Hw = sum(sum(x[i]*x[j] for i in range(n)) for j in range(n))
Hc = Constraint((sum(x[i] for i in range(1,n)) - 1) ** 2, label="onehot")
param_hw = Placeholder('hw')
param_hc = Placeholder("hc")
H = param_hw * Hw + param_hc * Hc

# Flip Option
onehot = [[f'x[{i}]' for i in range(1,n)]]
fixed = [['x[0]', 0]]
constflip = {"onehot" : onehot, "fixed":fixed}

# create model
model = H.compile()
feed_dict = {"hw":1, "hc":2}
qubo, offset = model.to_qubo(feed_dict=feed_dict)
z = Array.create('z', shape=3, vartype='BINARY')
high_order = {('z[0]', 'z[1]', 'z[2]'): 1, ('z[0]', 'z[1]', 'x[0]'):2}
spin_list = model.variables
spin_list = list(set(spin_list) |{'z[0]', 'z[1]', 'z[2]'})

# paramter_estimator
est = parameter_estimator.Estimator()
beta_range = est.get_beta_range(Hc, feed_dict,spin_list, qubo, num_sweeps=1000, high_order=high_order, **constflip)

#VA
sa_model = VectorAnnealing.model(qubo, offset, **constflip, high_order=high_order)
sampler = VectorAnnealing.sampler()
results = sampler.sample(sa_model, num_sweeps=1000, beta_range=beta_range, vector_mode=VectorAnnealing.VECTOR_MODE_SPEED)
```

Case 3

- This is a case where betas are estimated in constraint mode. Please follow the steps below for usage.
  - In this case, it is not possible to estimate weights (not required for solving in constraint mode).
  - In this case, it is possible to estimate a model that includes higher-order terms.

1. Importing the module
2. Creating the model
3. Creating the Estimator instance
4. Estimating betas
5. Executing annealing

Case 3 - Preprocessing

- Module import
  - Import parameter_estimator, pyqubo, and VectorAnnealing
    - Import the installed parameter_estimator.

```
import VectorAnnealing
from parameter_estimator  import  parameter_estimator
from pyqubo import Array, Placeholder, Constraint
```

Only vartype='BINARY' in Array.create is supported.
Others such as UnaryEncInteger and LogEncInteger are not supported.

Case 3 - Creating the Model

- Define the spin and set the objective function and constraints.
  - Define each constraint individually using Constraint().
  - Notes:
    - Labeling multiple constraints at once will not function correctly.
    - MIN(), MAX(), MINMAX(), and FREE() are reserved words and cannot be used as label names.

```
n = 5
x = Array.create('x', shape=n, vartype='BINARY')

# Define the objective function Hw
Hw = sum(sum(x[i]*x[j] for i in range(n)) for j in range(n))

# Define the objective function Hc
Hc = Constraint((sum(x[i] for i in range(n)) - 1) ** 2, label="onehot")

param_hw = Placeholder('hw')
param_hc = Placeholder("hc")
H = param_hw * Hw + param_hc * Hc
```

Case 3 - Creating the Model

- Setting Flip Options
  - Set each flip option according to the VA user's guide.
  - It is possible to set a flip option specific to constraint mode.
  - At least one flip option needs to be set.
  - For example, when using onehot, fixed and weighted_sum, set the flip options in constflip as shown below.

```
onehot = [[f'x[{i}]' for i in range(n)]]
fixed = [['x[0]', 0]]
weighted_sum = [{"comparison":("EQ", 3), "weight":{f'x[{i}]':(i+1) for i in range(n)}}]
constflip = {"onehot":onehot, "fixed":fixed, "weighted_sum":weighted_sum}
```

(Reference) List of flip options supported by Paramter_Estimator

  - If there are spins included only in the flip options, you need to add those spins to the spin_list before estimation.

Case 3 - Creating the Model

- Creating the model and qubo
  - Create the model and qubo from the created Hamiltonian.
  - Prepare feed_dict separately.

```
model = H.compile()
feed_dict = {"hw":2, "hc":1}
qubo, offset = model.to_qubo(feed_dict=feed_dict)
```

- Creating higher-order terms
  - When estimating only beta, it is also possible to specify higher-order terms. (Optional)

```
high_order = {('z[0]', 'z[1]', 'z[2]'):1, ('z[0]', 'z[1]', 'x[0]'):2}
```

  - If there are spins included only in the higher-order terms, you will need to add them to the spin_list later.

Case 3 - Creating the Model

- Creating the spin_list
  - Create the list of spins included in the model.
  - The list of spins used in the model created by PyQUBO is defined in model.variables. Please copy it.

```
spin_list = model.variables
```

  - If there are spins included only in the flip options, please add them manually.

```
y = Array.create('y', shape=3, vartype='BINARY')
high_order = {('z[0]', 'z[1]', 'z[2]'):1, ('z[0]', 'z[1]', 'x[0]'):2}
orone = ['x[0]', 'y[0]', 'y[1]', 'y[2]']
constflip["orone"] = [orone]
spin_list = list(set(spin_list) | set(orone) | {'z[0]', 'z[1]', 'z[2]'})#add y[0], y[1], y[2], z[0], z[1], z[2]
```

  - Including unnecessary spins may result in incorrect estimations.

Case 3 - Creating an Instance

- Creating an instance of the Estimator class
  - Create an instance as shown below.

```
#Creating an instance
est = parameter_estimator.Estimator()
```

  - Notes: If the Hamiltonian is changed, you will need to recreate the instance. If you do not recreate the instance, there is a possibility of incorrect estimation.

Case 3 – Estimating the Beta

- Automatic Estimation of Betas with the Parameter Estimator
  - Use the get_beta() function or the get_beta_range() function to automatically estimate betas.
  - H and feed are not required.
  - At least one flip option needs to be specified.
    - When estimating beta_range:

```
beta_range = est.get_constraint_beta_range(spin_list, qubo, num_sweeps=1000,
                                           high_order=high_order, **constflip)
```

    - When estimating beta_list:

```
beta_list = est.get_constraint_beta(spin_list, qubo, num_sweeps=1000, high_order=high_order,
**constflip)
```

Case 3 – Execution of VA

- Compute the va_model and input the va_model and the beta into va_sampler to perform the solution.
  - Make sure not to confuse beta_list with beta_range.
    - When estimating beta_list:

```
beta_list = est.get_constraint_beta(spin_list, qubo, num_sweeps=1000, high_order=high_order,
                                    **constflip)

va_model = VectorAnnealing.model(qubo, offset, high_order=high_order, **constflip)
va       = VectorAnnealing.sampler()
results = va.sample(va_model, beta_list=beta_list, num_sweeps=1000,
                            vector_mode=VectorAnnealing.VECTOR_MODE_CONSTRAINT)
```

    - When estimating beta_range:

```
beta_range = est.get_constraint_beta_range(spin_list, qubo, num_sweeps=1000,
                                           high_order=high_order,
                                           high_order=high_order, **constflip)

va_model = VectorAnnealing.model(qubo, offset, high_order=high_order, **constflip)
va       = VectorAnnealing.sampler()
results = va.sample(va_model, beta_range=beta_range, num_sweeps=1000,
                            vector_mode=VectorAnnealing.VECTOR_MODE_CONSTRAINT)
```

Case 3 - Sample Code

```
import VectorAnnealing
from parameter_estimator import parameter_estimator
from pyqubo import Array,Placeholder,Constraint

n= 5
x = Array.create('x', shape=n, vartype='BINARY')
Hw = sum(sum(x[i]*x[j] for i in range(n)) for j in range(n))
Hc = Constraint((sum(x[i] for i in range(n)) - 1) ** 2, label="onehot")
param_hw = Placeholder('hw')
param_hc = Placeholder("hc")
H = param_hw * Hw + param_hc * Hc

# Flip Option
onehot = [[f'x[{i}]' for i in range(n)]]
fixed = [['x[0]',0]]
weighted_sum = [{"comparison":("EQ",3), "weight":{f'x[{i}]':(i+1) for i in range(n)}}]
constflip = {"onehot":onehot, "fixed":fixed, "weighted_sum":weighted_sum}

# create model
model = H.compile()
feed_dict = {"hw":1, "hc":2}
qubo, offset = model.to_qubo(feed_dict=feed_dict)
z = Array.create('z', shape=3, vartype='BINARY')
high_order = {('z[0]', 'z[1]', 'z[2]'):1, ('z[0]', 'z[1]', 'x[0]'):2}
spin_list = model.variables
spin_list = list(set(spin_list) |{'z[0]', 'z[1]', 'z[2]'})

# paramter_estimator
est = parameter_estimator.Estimator()
beta_range = est.get_constraint_beta_range(spin_list, qubo, num_sweeps=1000, high_order=high_order, **constflip)

#VA
sa_model = VectorAnnealing.model(qubo, offset, **constflip, high_order=high_order)
sampler = VectorAnnealing.sampler()
results = sampler.sample(sa_model, num_sweeps=1000, beta_range=beta_range, vector_mode=VectorAnnealing.VECTOR_MODE_CONSTRAINT)
```

4. Specifications

The Estimator Class

- When using the Parameter Estimator, create an instance of the Estimator class.
- The following arguments can be set during the initialization of the Estimator instance.

Arguments	Required /Optional	Description
threads	Optional	Specify the number of threads for parallel computation. The default is half the number of logical cores (rounded up).

- To estimate parameters, execute the methods of the Estimator instance.

API specifications – get_feed()

- get_feed() accepts the following arguments.

Arguments	Required/Optional	Description
H	Required	Hamiltonian (In the case of omitting the objective function and specifying only the constraints, processing will be faster. In the case of having only the objective function, do not use get_feed because weight value will not be estimated.)
feed	Required	Values of manually prepared objective functions/constraints (positive values)
spin_list	Required	The list of spins used in H, flip option, and high_order. Specify PyQUBO's model.variables. If there are spins included only in the flip option, add those spins to PyQUBO's model.variables.
qubo	Required	QUBO output by the PyQUBO compilation function
**flip option	Optional	Various constraint flip options. The flip options that can be specified are described later. It is recommended to define it as a dictionary like constflip = {"onehot": [['x[0]', 'x[1]']], "fixed": [['x[0', 0]]} and specify it in a packed form with **constflip.
high_order	Optional	Note that get_feed() does not support this. Although the argument exists, specifying it will result in an error.
cal_all_feed	Optional	Whether to consider the objective function when calculating the weights. •If True, it calculates weights considering terms not marked with Constraint(). •If False, it calculates weights ignoring terms not marked with Constraint(). The default is False.

- The weight estimation function is not supported in constraint mode.
- The adjustment of weights is performed relatively by comparing multiple weights. Therefore, if cal_all_feed=False and only one weight for the constraint is considered, the weight cannot be adjusted, and it might be impossible to satisfy the constraints.

API specifications – get_beta()

- get_beta() accepts the following arguments.

Arguments	Required/Optional	Description
H	Required	Hamiltonian (In the case of omitting the objective function and specifying only the constraints, processing will be faster. In the case of having only the objective function, do not use get_feed because weight value will not be estimated.)
feed	Required	Values of manually prepared objective functions/constraints (positive values)
spin_list	Required	The list of spins used in H, flip option, and high_order. Specify PyQUBO's model.variables. If there are spins included only in the flip option, add those spins to PyQUBO's model.variables.
qubo	Required	QUBO output by the PyQUBO compilation function
sweeps	Optional	Number of sweeps in VectorAnnealing
**flip option	Optional	Various constraint flip options. The flip options that can be specified are described later. It is recommended to define it as a dictionary like constflip = {"onehot": [['x[0]', 'x[1]']], "fixed": [['x[0', 0]]} and specify it in a packed form with **constflip.
high_order	Optional	Higher-order term

API specifications - get_beta_range()

- get_beta_range() accepts the following arguments.

Arguments	Required/Optional	Description
H	Required	Hamiltonian (In the case of omitting the objective function and specifying only the constraints, processing will be faster. In the case of having only the objective function, do not use get_feed because weight value will not be estimated.)
feed	Required	Values of manually prepared objective functions/constraints (positive values)
spin_list	Required	The list of spins used in H, flip option, and high_order. Specify PyQUBO's model.variables. If there are spins included only in the flip option, add those spins to PyQUBO's model.variables.
qubo	Required	QUBO output by the PyQUBO compilation function
sweeps	Optional	Number of sweeps in VectorAnnealing
**flip option	Optional	Various constraint flip options. The flip options that can be specified are described later. It is recommended to define it as a dictionary like constflip = {"onehot": [['x[0]', 'x[1]']], "fixed": [['x[0', 0]]} and specify it in a packed form with **constflip.
high_order	Optional	Higher-order term
sweep_step_rate	Optional	The ratio of the number of steps in beta_range to the number of sweeps. The default is 0.8.

API specifications - get_ constraint_beta()

- get_constraint_beta() accepts the following arguments.

Arguments	Required/Optional	Description
spin_list	Required	The list of spins used in H, flip option, and high_order. Specify PyQUBO's model.variables. If there are spins included only in the flip option, add those spins to PyQUBO's model.variables.
qubo	Required	QUBO output by the PyQUBO compilation function
sweeps	Optional	Number of sweeps in VectorAnnealing
**flip option	Optional	Various constraint flip options. A flip option specific to constraint mode can also be specified. The flip options that can be specified are described later. It is recommended to define it as a dictionary like constflip = {"onehot": [['x[0]', 'x[1]']], "fixed": [['x[0', 0]]} and specify it in a packed form with **constflip.
high_order	Optional	Higher-order term

API specifications – get_constraint_beta_range()

- get_constraint_beta_range() accepts the following arguments.

Arguments	Required/Optional	Description
spin_list	Required	The list of spins used in H, flip option, and high_order. Specify PyQUBO's model.variables. If there are spins included only in the flip option, add those spins to PyQUBO's model.variables.
qubo	Required	QUBO output by the PyQUBO compilation function
sweeps	Optional	Number of sweeps in VectorAnnealing
**flip option	Optional	Various constraint flip options. A flip option specific to constraint mode can also be specified. The flip options that can be specified are described later. It is recommended to define it as a dictionary like constflip = {"onehot": [['x[0]', 'x[1]']], "fixed": [['x[0', 0]]} and specify it in a packed form with **constflip.
high_order	Optional	High-order term
sweep_step_rate	Optional	The ratio of the number of steps in beta_range to the number of sweeps. The default is 0.8.

flip option

- The flip options that can be specified are as follows.
  (For details, please refer to the VA user guide.)
  - The flip options that can be used constraint mode is enabled or not are as follows:

Arguments	Required /Optional	Description
onehot	Optional	Onehot constraint flip option
fixed	Optional	Fixed constraint flip option
andzero	Optional	Andzero constraint flip option
orone	Optional	Oreone constraint flip option
supplement	Optional	Supplement constraint flip option
maxone	Optional	Maxone constraint flip option
minmaxone	Optional	Minmaxone constraint flip option(If setting the condition, it is only for constraint mode)

  - The flip options that can only be used in constraint mode are as follows:

Arguments	Required /Optional	Description
minmaxone	Optional	Minmaxone constraint flip option(Condition can be set)
pattern	Optional	Pattern constraint flip option
weighted_sum	Optional	Weighted_sum constraint flip option

5. Notes for Use

Notes for Use - General

Note the following points

- For constraints, multiple constraint conditions must not be combined with one label.
  - Defining them collectively as follows will not yield correct estimation results.

```
Ha = param_time * Constraint(sum((sum(x[i, j] for i in range(point_num)) - 1)**2 for j in range(point_num)), label="onehot1")
```

- If a term with a coefficient of 0 is included in the equation, there may be instances where parameters different from the assumptions are estimated.

Notes for Use - General

- Note the following points
  - The operation cannot be guaranteed if the "nan" and "inf" values are used in the Hamiltonian and Placeholder. Be sure to avoid using them.
  - Please note that half of the x86 logical cores are used when using this tool.

Notes for Use - General

Note the following points

- In PyQUBO 1.4.0, the following specification changes have been observed. Please be cautious when using code created with older versions.
  - If there is a sum of terms with and without the product of Placeholders, an error will occur with "TypeError: **radd**(): incompatible function arguments." Therefore, please make corrections as shown in the example below.
    - Example that results in an error:

```
HW += sum((dlength[j][k] - d_min[j]) * x[i, j] * x[(i+1)%point_num, k] for i in range(point_num))
H = param_hw * HW + HC
```

    - Modified example:

```
HW += param_hw * sum(, high_order=high_order,  (dlength[j][k] - d_min[j]) * x[i, j] * x[(i+1)%point_num, k] for i in range(point_num))
H = HW + HC
```

Notes for Use - General

- Note the following points
  - The execution time of Parameter Estimator vary depend on a given program.
    - Constraint item (feed_dict): feed1, feed2
    - Constraint equation (Constraint function): const1, const2, const3

① Example where execution time is faster

```
HC1 = feed1*(const1 + const2 + const3)
HC2 = feed2*(const4 + const5 + const6)
HC = HC1 + HC2
```

② Example where execution time is slower

```
HC1 = feed1*const1 + feed1*const2 + feed1*const3
HC2 = feed2*const4 + feed2*const5 + feed2*const6
HC = HC1 + HC2
```

- With method (1) you can shorten the processing time due to the ability to parallelize the addition of the same feed into each const with multiple CPU cores. Because (const1+const2+const3) can be computed in parallel.
- With method (2) you can only compute with one CPU core and cannot use parallelization, because the feeds are added into const one by one. Because feed1*const1, feed1*const2... are each one parallel operation.

It is recommended that you formulate the constraint equation using method (1).

Notes for Use - General

- Note the following points
  - About the Constraint() label specification
    - In the following cases, a specification which uses a reserved word in a label is required.

Conditions	Reserved words
Defining a constraint with a range specification in Constraint	MIN(), MAX(), MINMAX()
Defining a constraint with an auxiliary spin in Constraint	FREE()

Notes for Use - General

- Note the following points
  - Defining a constraint with a range specification in Constraint
    - Some problems require a **range specification of the total number of spins of 1** in the Constraint expression when estimating beta; specifying MIN() or MAX() for label allows Constraint() values to be non-zero.
    - For example, the following is a sample code for the shift problem which provides a formula for approaching the target value for the number of employees as an example. Set the minimum and maximum for the total number of spins in 1 in the label MIN({}) and MAX({}).
    - If the minimum and maximum values are the same, it is also possible to set them as MINMAX({}).

```
target_n = TARGET_WORK / float(D)  #Target value
energy_min = (int(target_n - 0.5) - target_n) ** 2      # Minimum number of spins in 1
energy_min = (int(target_n - 1.5) - target_n) ** 2      # Maximum number of spins in 1

Hc = (sum(Constraint((sum(spin_x[i,j] for i in range(N)) - target_n) ** 2,
                     label="employeers_in_day_{}_MIN({})_MAX({})".format(j, energy_min, energy_max))
          for j in range(D)) / (scale_n * D))
```

Notes for Use - General

- Note the following points
  - Defining a constraint with an auxiliary spin in Constraint
    - By specifying FREE() for the label, it is possible to avoid a constraint violation in the case where the specified spin is flipped and satisfies the constraints even when the Constraint() value is non-zero.
    - When there are multiple auxiliary spins, add them in the form of "H1_FREE({})_FREE({})" etc.

```
H1_1 = y[0] * x[2]
H1_2 = 3 * y[0] + x[0] * x[1] – 2 * y[0] * x[0] – 2 * y[0] * x[1]
H1 += Constraint(H1_1 + H1_2, label="H1_FREE({})".format(y[0]))
```

Notes for Use - General

- Note the following points
  - When there are many constraints with multiple "Constraint()" in user programs, a segmentation violation will occur during pyqubo::compile() execution (*1).
    - When there are many constraints, separate the variables between the formula used in the pyqubo::compile() input and the formula which is input to the arguments of the Parameter Estimator.

```
H = 0              # Variable for use by the parameter_estimator
H_compile = 0 # Variable for use in compiling
for k in range(N):
    for num,(i,j) in enumerate(M):
        # Constraint is used for the formula input as an argument for the parameter_estimator
        H  += Constraint(x[i,k] * x[j,k], f"H_{k}_{num}")
        # Constraint is removed for the formula used for the pyqubo::compile() input
        H_compile += x[i,k] * x[j,k]
. . .
# Variables for compilation are used for compilation
Model = H_compile.compile()
. . .
# When parameter_estimator is used, variables which use Constraint are used
Betas = est.get_beta(H, feed_dict, spin_list, qubo, sweeps, **constflip)
```

*1: Verifying events in cases where 600,000 or more constraints exist

Notes for Use - General

- Note the following points
  - When running problems with a massive problem size or cohesion density, you may run into low memory conditions and be unable to run get_feed()/get_beta()/get_beta_range() in some cases.
    - C++ (Pybind) low memory error
      - The following is a case which occurs when the required amount of memory cannot be secured during Parameter Estimator cpp side processing.

```
terminate called without an active exception
```

    - Low memory error on the Python side
      - The following is also a case which occurs when the required amount of memory cannot be secured in Parameter Estimator Python side processing.

```
OSError: [Errno 12] Cannot allocate memory
```

      - Currently, it is possible to operate by deleting the Parameter Estimator instance called with get_beta and then avoiding the memory leak with gc.collect().

Notes for Use - Beta

- Note the following points
  - Either beta_list or beta_range should be used. It is difficult to say which will be more accurate, as it depends on the problem. It is recommended to try beta_list first, and if the accuracy is not good enough, try beta_range.
  - The first value in the beta_list estimated with get_beta() and the first value estimated with get_beta_range() do not necessarily match.
  - The return values of get_beta() and get_beta_range() are both of type list; if you mistakenly specify the beta_range estimated by get_beta_range() in the beta_list, it will work as the unexpected beta. Please be careful which beta you are using.

```
NG example)
beta_range = est.get_beta_range(HC, feed_dict, spin_list, qubo, sweeps, **constflip)
results = va.sample(va_model, sweeps, beta_list=beta_range)
```

Notes for Use - Weight

- Note the following points
  - If an expression with no constraints is specified, no estimation is performed, and the initial feed entered is returned as is.
  - The feed_dict used to create the qubo file and the feed_dict specified as the argument to get_feed() should be the same.
  - Please set the weights for the objective function as well.
  - In the default setting, terms other than the constraints are ignored. If you want the estimation to take the objective function into account, specify cal_all_flip=True.

Issuance History

- Issuance history list

July	2022	1st Edition
November	2022	2nd Edition
November	2023	3rd Edition
January	2024	4th Edition
November	2024	5th Edition
November	2025	6th Edition

- Details of additions and changes

1st Edition	New
2nd Edition	Added precautions for version of PyQUBO
3rd Edition	Changes due to addition of get_bata_range() and get_feed()
4th Edition	Remove the description of Vector Engine.
5th Edition	Changes due to support for constraint mode/high-order and documentation adjustments
6th Edition	Changes due to fix some sample codes

NEC Vector Annealing Parameter Estimator User's Guide

November 2025, 6th Edition

NEC Corporation
7-1, Shiba 5-chome, Minato-ku, Tokyo
TEL (03) 3454-1111 (Main)


Orchestrating a brighter world
NEC