pyqubo Documentation

Release 1.0.5

Author

Jan 21, 2021

Manual:

PyQUBO allows you to create QUBOs or Ising models from flexible mathematical expressions easily. Some of the features of PyQUBO are

- **Python based (C++ backend).**
- **Fully integrated with Ocean SDK.** (details)
- **Automatic validation of constraints.** (details)
- **Placeholder** for parameter tuning. (details)

For more details, see PyQUBO Documentation.

CHAPTER 1

Example Usage

1.1 Creating QUBO

This example constructs a simple expression and compile it to `model`. By calling `model.to_qubo()`, we get the resulting QUBO. (This example solves Number Partitioning Problem with a set S = {4, 2, 7, 1})

```
>>> from pyqubo import Spin
>>> s1, s2, s3, s4 = Spin("s1"), Spin("s2"), Spin("s3"), Spin("s4")
>>> H = (4*s1 + 2*s2 + 7*s3 + s4)**2
>>> model = H.compile()
>>> qubo, offset = model.to_qubo()
>>> pprint(qubo)
{('s1', 's1'): -160.0,
('s1', 's2'): 64.0,
('s2', 's2'): -96.0,
('s3', 's1'): 224.0,
('s3', 's2'): 112.0,
('s3', 's3'): -196.0,
('s4', 's1'): 32.0,
('s4', 's2'): 16.0,
('s4', 's3'): 56.0,
('s4', 's4'): -52.0}
```

1.2 Integration with D-Wave Ocean

PyQUBO can output the BinaryQuadraticModel(BQM) which is compatible with *Sampler* class defined in D-Wave Ocean SDK. In the example below, we solve the problem with *SimulatedAnnealingSampler*.

```
>>> import neal
>>> sampler = neal.SimulatedAnnealingSampler()
>>> bqm = model.to_bqm()
```

(continues on next page)

(continued from previous page)

```
>>> sampleset = sampler.sample(bqm, num_reads=10)
>>> decoded_samples = model.decode_sampleset(sampleset)
>>> best_sample = min(decoded_samples, key=lambda x: x.energy)
>>> best_sample.sample # doctest: +SKIP
{'s1': 0, 's2': 0, 's3': 1, 's4': 0}
```

If you want to solve the problem by actual D-Wave machines, just replace the *sampler* by a *DWaveCliqueSampler* instance, for example.

For more examples, see example notebooks.

CHAPTER 2

Benchmarking

Since the core logic of the new PyQUBO (>=1.0.0) is written in C++ and the logic itself is also optimized, the execution time to produce QUBO has become shorter. We benchmarked the execution time to produce QUBOs of TSP with the new PyQUBO (1.0.0) and the previous PyQUBO (0.4.0). The result shows the new PyQUBO runs 1000 times faster as the problem size increases.

Execution time includes building Hamiltonian, compilation, and producing QUBOs. The code to produce the above result is found in here.

CHAPTER 3

Installation

```
pip install pyqubo
```

or

```
python setup.py install
```

CHAPTER 4

Supported Python Versions

Python 3.5, 3.6, 3.7, 3.8 and 3.9 are supported.

CHAPTER 5

Supported Operating Systems

- Linux (32/64bit)
- OSX (64bit, >=10.9)
- Win (64bit)

5.1 Getting Started

5.1.1 Installation

If you use pip, just type

```
pip install pyqubo
```

You can install from the source code like

```
git clone https://github.com/recruit-communications/pyqubo.git
cd pyqubo
python setup.py install
```

5.1.2 QUBO and Ising Model

If you want to solve a combinatorial optimization problem by quantum or classical annealing machines, you need to represent your problem in QUBO (Quadratic Unconstrained Binary Optimization) or Ising model. PyQUBO converts your problem into QUBO or Ising model format.

The objective function of **QUBO** is defined as:

$$\sum_{i \leq j} q_{ij} x_i x_j$$

where x_i represents a binary variable which takes 0 or 1, and q_{ij} represents a quadratic coefficient. Note that $q_{ii}x_ix_i = q_{ii}x_i$, since $x_i^2 = x_i$. Thus, the above expression includes linear terms of x_i.

The objective function of **Ising model** is defined as:

$$\sum_i h_i s_i + \sum_{i<j} J_{ij} s_i s_j$$

where s_i represents spin variable which takes -1 or 1, h_i represents an external magnetic field and J_{ij} represents an interaction between spin i and j.

5.1.3 Basic Usage

With PyQUBO, you can construct QUBOs with 3 steps:

1. **Define the Hamiltonian.**

```
>>> from pyqubo import Spin
>>> s1, s2, s3, s4 = Spin("s1"), Spin("s2"), Spin("s3"), Spin("s4")
>>> H = (4*s1 + 2*s2 + 7*s3 + s4)**2
```

2. **Compile the Hamiltonian to get a model.**

```
>>> model = H.compile()
```

3. **Call 'to_qubo()' to get QUBO coefficients.**

```
>>> qubo, offset = model.to_qubo()
>>> pprint(qubo) # doctest: +SKIP
{('s1', 's1'): -160.0,
 ('s1', 's2'): 64.0,
 ('s1', 's3'): 224.0,
 ('s1', 's4'): 32.0,
 ('s2', 's2'): -96.0,
 ('s2', 's3'): 112.0,
 ('s2', 's4'): 16.0,
 ('s3', 's3'): -196.0,
 ('s3', 's4'): 56.0,
 ('s4', 's4'): -52.0}
>>> print(offset)
196.0
```

In this example, you want to solve Number Partitioning Problem with a set S = {4, 2, 7, 1}. The hamiltonian H is represented as

$$H = (4s_1 + 2s_2 + 7s_3 + s_4)^2$$

where s_i is a i th spin variable which indicates a group the i th number should belong to. In PyQUBO, spin variables are internally converted to binary variables via the relationship $x_i = (s_i + 1)/2$. The QUBO coefficents and the offset returned from `Model.to_qubo()` represents the following objective function:

$$\begin{aligned} &-160x_1x_1 + 64x_1x_2 + 224x_1x_3 + 32x_1x_4 - 96x_2x_2 \\ &+ 112x_2x_3 + 16x_2x_4 - 196x_3x_3 + 56x_3x_4 - 52x_4x_4 + 196 \end{aligned}$$

4. **Call 'to_ising()' to get Ising coefficients.**

If you want to get the coefficient of the Ising model, just call `to_ising()` method like below.

```
>>> linear, quadratic, offset = model.to_ising()
>>> pprint(linear) # doctest: +SKIP
{'s1': 0.0, 's2': 0.0, 's3': 0.0, 's4': 0.0}
>>> pprint(quadratic) # doctest: +SKIP
{('s1', 's2'): 16.0,
 ('s1', 's3'): 56.0,
 ('s1', 's4'): 8.0,
 ('s2', 's3'): 28.0,
 ('s2', 's4'): 4.0,
 ('s3', 's4'): 14.0}
>>> print(offset)
70.0
```

where *linear* represents external magnetic fields h, *quadratic* represents interactions J and *offset* represents the constant value in the objective function below.

$$16s_1s_2 + 56s_1s_3 + 8s_1s_4 + 28s_2s_3 + 4s_2s_4 + 14s_3s_4 + 70$$

5.1.4 Variable: Binary and Spin

When you define a Hamiltonian, you can use `Binary` or `Spin` class to represent $\{0, 1\}$ or $\{1, -1\}$ variable.

Example: If you want to define a Hamiltonian with binary variables $x \in \{0, 1\}$, use `Binary`.

```
>>> from pyqubo import Binary
>>> x1, x2 = Binary('x1'), Binary('x2')
>>> H = 2*x1*x2 + 3*x1
>>> pprint(H.compile().to_qubo()) # doctest: +SKIP
({('x1', 'x1'): 3.0, ('x1', 'x2'): 2.0, ('x2', 'x2'): 0.0}, 0.0)
```

Example: If you want to define a Hamiltonian with spin variables $s \in \{-1, 1\}$, use `Spin`.

```
>>> from pyqubo import Spin
>>> s1, s2 = Spin('s1'), Spin('s2')
>>> H = 2*s1*s2 + 3*s1
>>> pprint(H.compile().to_qubo()) # doctest: +SKIP
({('s1', 's1'): 2.0, ('s1', 's2'): 8.0, ('s2', 's2'): -4.0}, -1.0)
```

5.1.5 Solve QUBO by dimod Sampler

PyQUBO model can output the BinaryQuadraticModel(BQM). You can solve BQM by using `Sampler` class. `Sampler` is an abstract class defined by dimod package. Various kinds of sampler class, such as SimulatedAnnealingSampler or DWaveSampler, inherits *Sampler* class.

First, we craete BQM object using `to_bqm()` method. (If you want to use DWaveSampler which only takes integer-indexed QUBO, you can simply do like `to_bqm(index_label=True)`.)

```
>>> from pyqubo import Binary
>>> x1, x2 = Binary('x1'), Binary('x2')
>>> H = (x1 + x2 - 1)**2
>>> model = H.compile()
>>> bqm = model.to_bqm()
```

Next, we create `neal.SimulatedAnnealingSampler` and use `sample()` method to get the solutions of QUBO as SampleSet. You can use `Model.decode_sampleset()` to interpret the *sampleset* object, and it returns *decoded_samples* which is a list of *`pyqubo.DecodedSample`* object.

```
>>> import neal
>>> sa = neal.SimulatedAnnealingSampler()
>>> sampleset = sa.sample(bqm, num_reads=10)
>>> decoded_samples = model.decode_sampleset(sampleset)
>>> best_sample = min(decoded_samples, key=lambda x: x.energy)
>>> pprint(best_sample.sample)
{'x1': 0, 'x2': 1}
```

5.1.6 Array of Variables

`Array` class represents a multi-dimensional array of `Binary` or `Spin`.

Example: You can access each element of the matrix with an index like:

```
>>> from pyqubo import Array
>>> x = Array.create('x', shape=(2, 3), vartype='BINARY')
>>> x[0, 1] + x[1, 2]
(Binary(x[0][1])+Binary(x[1][2]))
```

Example: You can use `Array` to represent multiple spins in the example of partitioning problem above.

```
>>> from pyqubo import Array
>>> numbers = [4, 2, 7, 1]
>>> s = Array.create('s', shape=4, vartype='SPIN')
>>> H = sum(n * s for s, n in zip(s, numbers))**2
>>> model = H.compile()
>>> qubo, offset = model.to_qubo()
>>> pprint(qubo) # doctest: +SKIP
{('s[0]', 's[0]'): -160.0,
 ('s[0]', 's[1]'): 64.0,
 ('s[0]', 's[2]'): 224.0,
 ('s[0]', 's[3]'): 32.0,
 ('s[1]', 's[1]'): -96.0,
 ('s[1]', 's[2]'): 112.0,
 ('s[1]', 's[3]'): 16.0,
 ('s[2]', 's[2]'): -196.0,
 ('s[2]', 's[3]'): 56.0,
 ('s[3]', 's[3]'): -52.0}
```

5.1.7 Placeholder

If you have a parameter that you will probably update, such as the strength of the constraints in your hamiltonian, using `Placeholder` will save your time. If you define the parameter by `Placeholder`, you can specify the value of the parameter after compile. This means that you don't have to compile repeatedly for getting QUBOs with various parameter values. It takes longer time to execute a compile when the problem size is bigger. In that case, you can save your time by using `Placeholder`.

Example: If you have an objective function $2a + b$, and a constraint $a + b = 1$ whose hamiltonian is $(a + b - 1)^2$ where a, b is qbit variable, you need to find the penalty strength M such that the constraint is satisfied. Thus, you need to create QUBO with different values of M. In this example, we create QUBO with $M = 5.0$ and $M = 6.0$.

In the first code, we don't use placeholder. In this case, you need to compile the hamiltonian twice to get a QUBO with $M = 5.0$ and $M = 6.0$.

```
>>> from pyqubo import Binary
>>> a, b = Binary('a'), Binary('b')
>>> M = 5.0
>>> H = 2*a + b + M*(a+b-1)**2
>>> model = H.compile()
>>> qubo, offset = model.to_qubo() # QUBO with M=5.0
>>> M = 6.0
>>> H = 2*a + b + M*(a+b-1)**2
>>> model = H.compile()
>>> qubo, offset = model.to_qubo() # QUBO with M=6.0
```

If you don't want to compile twice, define M by `Placeholder`.

```
>>> from pyqubo import Placeholder
>>> a, b = Binary('a'), Binary('b')
>>> M = Placeholder('M')
>>> H = 2*a + b + M*(a+b-1)**2
>>> model = H.compile()
>>> qubo, offset = model.to_qubo(feed_dict={'M': 5.0})
```

You get a QUBO with different value of M without compile

```
>>> qubo, offset = model.to_qubo(feed_dict={'M': 6.0})
```

The actual value of the placeholder M is specified in calling `Model.to_qubo()` as a value of the feed_dict.

5.1.8 Validation of Constraints

When you get a solution from the Sampler, `Model.decode_sample()` decodes the sample and returns `DecodedSample` object.

Example: You are solving a partitioning problem.

```
>>> from pyqubo import Binary, Constraint
>>> a, b = Binary('a'), Binary('b')
>>> M = 5.0 # strength of the constraint
>>> H = 2*a + b + M * Constraint((a+b-1)**2, label='a+b=1')
>>> model = H.compile()
```

Let's assume that you get a solution `{'a': 0, 'b': 1}` from the solver.

```
>>> raw_solution = {'a': 0, 'b': 1} # solution from the solver
>>> decoded_sample = model.decode_sample(raw_solution, vartype='BINARY')
>>> pprint(decoded_sample.sample)
{'a': 0, 'b': 1}
>>> pprint(decoded_sample.constraints())
{'a+b=1': (True, 0.0)}
>>> pprint(decoded_sample.constraints(only_broken=True))
{}
```

You can access to the dict of the sample via `decoded_sample.sample`. You can also access to the value of the constraint of the Hamiltonian via *decoded_sample.constraints()*. If you specify the argument *only_broken=True*, only broken constraint will be returned. If the empty *dict* is returned, it indicates that there is no broken constraint corresponding to the given sample.

5.2 Contribution

Thank you for contributing to PyQUBO.

Propose a new feature and implement

1. If you have a proposal of new features, send a pull request with your idea and we will discuss it.
2. Once we agree with the new feature, implement the feature. If you implement a new module on top of PyQUBO, create your module inside the `pyqubo/contrib` directory.

Implement a feature or bug-fix for an existing issue

1. See the issue list of github.
2. Choose an issue and comment on the task that you will work on.
3. Send a pull request.

Implementing unittests for your feature helps a review process.

5.2.1 Installation

If you already installed PyQUBO, uninstall it.

```
pip uninstall pyqubo
```

Install PyQUBO with development mode

```
python setup.py develop
```

5.2.2 Coding Conventions

- Follow PEP8.
- Write docstring with Google docstrings convention.
- Write unit tests.
- Write comments when the code is complicated. But the best documentation is clean code with good variable names.

5.2.3 Unit Testing

To run unit tests, you have two options. One option is to run with unittest or coverage command. To run all tests with unittest, execute

```
python -m unittest discover tests
```

To generate coverage reports, execute

```
coverage run -m unittest discover
coverage html
```

You will see html files of the report in `htmlcov` directory.

Second option is to run test using docker container with circleci CLI locally. To run test with circleci CLI, execute

```
circleci build --job $JOBNAME
```

$JOBNAME needs to be replaced with a job name such as *test-3.6*, listed in `.circleci/config.yml`. To install circleci CLI, refer to https://circleci.com/docs/2.0/local-cli/.

5.2.4 Documentation

Documents are created by sphinx from the docstring in Python code. When you add a new class, please create a new rst file in `docs/reference` directory. If the information of the class is not important for library users, create a file under `internal` directory. To build html files of document locally, execute

```
make clean html
```

You can see built htmls in `docs/_build` directory. When you write an example code in docstring, you can test the code with doctest. To run doctest, execute

```
make doctest
```

5.3 Expression

class Base

Abstract class of pyqubo expression.

All basic component class such as *Binary*, *Spin* or *Add* inherits *Base*.

For example, an expression $2ab + 1$ (where a, b is *Binary* variable) is represented by the binary tree above.

Note: This class is an abstract class of all component of expressions.

Example:

We write mathematical expressions with objects such as *Binary* or *Spin* which inherit *Base*.

```
>>> from pyqubo import Binary
>>> a, b = Binary("a"), Binary("b")
>>> 2*a*b + 1
(Binary(a)*Num(2.000000)*Binary(b)+Num(1.000000))
```

compile(*strength=5.0*)

Returns the compiled *Model*.

This method reduces the degree of the expression if the degree is higher than 2, and convert it into *Model* which has information about QUBO.

Parameters **strength** (*float*) – The strength of the reduction constraint. Insufficient strength can result in the binary quadratic model not having the same minimizations as the polynomial.

Returns The model compiled from the *Base*.

Return type *Model*

Examples:

In this example, there are higher order terms abc and abd. It is decomposed as [[a*b, c], d] hierarchically and converted into QUBO. By calling *to_qubo()* of the `model`, we get the QUBO.

```
>>> from pyqubo import Binary
>>> a, b, c, d = Binary("a"), Binary("b"), Binary("c"), Binary("d")
>>> model = (a*b*c + a*b*d).compile()
>>> pprint(model.to_qubo()) # doctest: +SKIP
({('a', 'a'): 0.0,
  ('a', 'a*b'): -10.0,
  ('a', 'b'): 5.0,
  ('a*b', 'a*b'): 15.0,
  ('a*b', 'b'): -10.0,
  ('a*b', 'c'): 1.0,
  ('a*b', 'd'): 1.0,
  ('b', 'b'): 0.0,
  ('c', 'c'): 0,
  ('d', 'd'): 0},
 0.0)
```

5.3.1 Binary

class Binary(*label*)

Binary variable i.e. {0, 1}.

Parameters **label** (*str*) – The label of a variable. A variable is identified by this label.

Example:

Example code to create an expression.

```
>>> from pyqubo import Binary
>>> a, b = Binary('a'), Binary('b')
>>> exp = 2*a*b + 3*a
>>> pprint(exp.compile().to_qubo())    # doctest: +SKIP
({('a', 'a'): 3.0, ('a', 'b'): 2.0, ('b', 'b'): 0}, 0.0)
```

5.3.2 Spin

class Spin(*label*)

Spin variable i.e. {-1, 1}.

Parameters **label** (*str*) – The label of a variable. A variable is identified by this label.

Example:

Example code to create an expression.

```
>>> from pyqubo import Spin
>>> a, b = Spin('a'), Spin('b')
>>> exp = 2*a*b + 3*a
>>> pprint(exp.compile().to_qubo()) # doctest: +SKIP
({('a', 'a'): 2.0, ('a', 'b'): 8.0, ('b', 'b'): -4.0}, -1.0)
```

5.3.3 Placeholder

class Placeholder(*label*)

Placeholder expression.

You can specify the value of the *Placeholder* when creating the QUBO. By using *Placeholder*, you can change the value without compiling again. This is useful when you need to update the strength of constraint gradually.

Parameters **label** (*str*) – The label of the placeholder.

Example:

The value of the placeholder is specified when you call *to_qubo()*.

```
>>> from pyqubo import Binary, Placeholder
>>> x, y, a = Binary('x'), Binary('y'), Placeholder('a')
>>> exp = a*x*y + 2.0*x
>>> pprint(exp.compile().to_qubo(feed_dict={'a': 3.0})) # doctest: +SKIP
({('x', 'x'): 2.0, ('x', 'y'): 3.0, ('y', 'y'): 0}, 0.0)
>>> pprint(exp.compile().to_qubo(feed_dict={'a': 5.0})) # doctest: +SKIP
({('x', 'x'): 2.0, ('x', 'y'): 5.0, ('y', 'y'): 0}, 0.0)
```

5.3.4 SubH

class SubH(*hamiltonian*, *label*, *as_constraint=False*)

SubH expression. The parent class of Constraint. You can specify smaller sub-hamiltonians in your expression.

Parameters

- **hamiltonian** (Base) – The expression you want to specify as a sub-hamiltonian.
- **label** (*str*) – The label of the sub-hamiltonian. Sub-hamiltonians can be identified by their labels.
- **as_constraint** (*boolean*) – Whether or not the sub-hamiltonian should also be treated as a constraint. False by default.

Example:

You can call namespaces to identify the labels defined in a model.

```
>>> from pyqubo import Spin, SubH
>>> s1, s2, s3 = Spin('s1'), Spin('s2'), Spin('s3')
>>> exp = (SubH(s1 + s2, 'n1'))**2 + (SubH(s1 + s3, 'n2'))**2
>>> model = exp.compile()
>>> model.namespaces  #doctest: +SKIP
({'n1': {'s1', 's2'}, 'n2': {'s1', 's3'}}, {'s1', 's2', 's3'})
```

5.3.5 Constraint

class Constraint(*hamiltonian*, *label*, *condition=lambda x: x==0.0*)

Constraint expression. You can specify the constraint part in your expression.

Parameters

- **child** (*Express*) – The expression you want to specify as a constraint.
- **label** (*str*) – The label of the constraint. You can identify constraints by the label.
- **(float => boolean) condition** (*func*) – function to indicate whether the constraint is satisfied or not. Default is *lambda x: x == 0.0*. function takes float value and returns boolean value. You can define the condition where the constraint is satisfied.

Example:

When the Hamiltonian contains *Constraint*, you know whether each constraint is satisfied or not by accessing to *DecodedSample*.

```
>>> from pyqubo import Binary, Constraint
>>> a, b = Binary('a'), Binary('b')
>>> H = Constraint(a+b-2, "const1") + Constraint(a+b-1, "const2")
>>> model = H.compile()
>>> dec = model.decode_sample({'a': 1, 'b': 0}, vartype='BINARY')
>>> pprint(dec.constraints())
{'const1': (False, -1.0), 'const2': (True, 0.0)}
>>> pprint(dec.constraints(only_broken=True))
{'const1': (False, -1.0)}
```

5.3.6 Add

class Add(*left*, *right*)

Addition of expressions.

Parameters

- **left** (Base) – An expression
- **right** (Base) – An expression

Example:

You can add expressions with either the built-in operator or *Add*.

```
>>> from pyqubo import Binary, Add
>>> a, b = Binary('a'), Binary('b')
>>> a + b
(Binary(a)+Binary(b))
>>> Add(a, b)
(Binary(a)+Binary(b))
```

5.3.7 Mul

class Mul(*left*, *right*)

Product of expressions.

Parameters

- **left** (Base) – An expression
- **right** (Base) – An expression

Example:

You can multiply expressions with either the built-in operator or *Mul*.

```
>>> from pyqubo import Binary, Mul
>>> a, b = Binary('a'), Binary('b')
>>> a * b
Binary(a)*Binary(b)
>>> Mul(a, b)
Binary(a)*Binary(b)
```

5.3.8 Num

class Num(*value*)
Expression of number

Parameters **value** (*float*) – the value of the number.

Example:

Example code to create an expression.

```
>>> from pyqubo import Binary, Num
>>> a = Binary('a')
>>> a + 1
(Binary(a)+Num(1.000000))
>>> a + Num(1)
(Binary(a)+Num(1.000000))
```

5.3.9 UserDefinedExpress

class UserDefinedExpress
User defined express.

User can define their own expression by inheriting *UserDefinedExpress*.

Example:

Define the LogicalAnd class by inheriting *UserDefinedExpress*.

```
>>> from pyqubo import UserDefinedExpress, Binary
>>> class LogicalAnd(UserDefinedExpress):
...     def __init__(self, bit_a, bit_b):
...         express = bit_a * bit_b
...         super().__init__(express)
>>> a, b = Binary('a'), Binary('b')
>>> logical_and = LogicalAnd(a, b)
```

5.3.10 WithPenalty

class WithPenalty
You can define the custum penalty class by inheriting *WithPenalty*. The *penalty* argument will be added to the generated Hamiltonian. Integer classes with constraints, such as *OneHotEncInteger*, are defined using this class.

Example:

Define the custom penalty class inheriting *WithPenalty*. We initialize this class with *hamiltonian* h. The constraint term $(h-1)^2$ will be added to the generated Hamiltonian.

```
>>> from pyqubo import WithPenalty
>>> class CustomPenalty(WithPenalty):
...     def __init__(self, hamiltonian, label, strength):
...         penalty = strength * (hamiltonian-1)**2
...         super().__init__(hamiltonian, penalty, label)
>>> a, b = Binary("a"), Binary("b")
>>> p = CustomPenalty(a+b, label="penalty", strength=2.0)
```

(continues on next page)

(continued from previous page)

```
>>> model = (p+1).compile()
>>> qubo, offset = model.to_qubo()
```

5.4 Model

5.4.1 Model

class Model

Model represents binary quadratic optimization problem.

By compiling `Express` object, you get a *Model* object. It contains the information about QUBO (or equivalent Ising Model), and it also has the function to decode the solution into the original variable structure.

Note: We do not need to create this object directly. Instead, we get this by compiling *Express* objects.

Generate QUBO, Ising model, and BQM

to_qubo()	Returns QUBO and energy offset.
to_ising()	Returns Ising Model and energy offset.
to_bqm()	Returns `dimod.BinaryQuadraticModel`.

Interpret samples returned from solvers

energy()	Returns energy of the sample.
decode_sample()	Returns Ising Model and energy offset.
decode_sampleset()	Decode the sample represented by `dimod.SampleSet`.

to_qubo(*index_label=False*, *feed_dict=None*)

Returns QUBO and energy offset.

Parameters

- **index_label** (*bool*) – If true, the keys of returned QUBO are indexed with a positive integer number.
- **feed_dict** (*dict[str,float]*) – If the expression contains *Placeholder* objects, you have to specify the value of them by *Placeholder*. Please refer to *Placeholder* for more details.

Returns Tuple of QUBO and energy offset. QUBO takes the form of `dict[(str, str), float]`.

Return type `tuple[QUBO, float]`

Examples:

This example creates the `model` from the expression, and we get the resulting QUBO by calling `model.to_qubo()`.

```
>>> from pyqubo import Binary
>>> x, y, z = Binary("x"), Binary("y"), Binary("z")
>>> model = (x*y + y*z + 3*z).compile()
```

(continues on next page)

(continued from previous page)

```
>>> pprint(model.to_qubo()) # doctest: +SKIP
({('x', 'x'): 0.0,
('x', 'y'): 1.0,
('y', 'y'): 0.0,
('z', 'y'): 1.0,
('z', 'z'): 3.0},
0.0)
```

If you want a QUBO which has index labels, specify the argument `index_label=True`. The mapping of the indices and the corresponding labels is stored in `model.variables`.

```
>>> pprint(model.to_qubo(index_label=True)) # doctest: +SKIP
({(0, 0): 3.0, (0, 2): 1.0, (1, 1): 0.0, (1, 2): 1.0, (2, 2): 0.0}, 0.0)
>>> model.variables
['z', 'x', 'y']
```

to_ising(*index_label=False*, *feed_dict=None*)

Returns Ising Model and energy offset.

Parameters

- **index_label** (*bool*) – If true, the keys of returned QUBO are indexed with a positive integer number.
- **feed_dict** (*dict[str, float]*) – If the expression contains *Placeholder* objects, you have to specify the value of them by *Placeholder*. Please refer to *Placeholder* for more details.

Returns Tuple of Ising Model and energy offset. Where *linear* takes the form of `(dict[str, float])`, and *quadratic* takes the form of `dict[(str, str), float]`.

Return type `tuple(linear, quadratic, float)`

Examples:

This example creates the `model` from the expression, and we get the resulting Ising model by calling *to_ising()*.

```
>>> from pyqubo import Binary
>>> x, y, z = Binary("x"), Binary("y"), Binary("z")
>>> model = (x*y + y*z + 3*z).compile()
>>> pprint(model.to_ising()) # doctest: +SKIP
({'x': 0.25, 'y': 0.5, 'z': 1.75}, {('x', 'y'): 0.25, ('z', 'y'): 0.25},␣
↪2.0)
```

If you want a Ising model which has index labels, specify the argument `index_label=True`. The mapping of the indices and the corresponding labels is stored in `model.variables`.

```
>>> pprint(model.to_ising(index_label=True)) # doctest: +SKIP
({0: 1.75, 1: 0.25, 2: 0.5}, {(0, 2): 0.25, (1, 2): 0.25}, 2.0)
>>> model.variables
['z', 'x', 'y']
```

to_bqm(*index_label=False*, *feed_dict=None*)

Returns `dimod.BinaryQuadraticModel`.

For more details about `dimod.BinaryQuadraticModel`, see dimod.BinaryQuadraticModel.

Parameters

- **index_label** (*bool*) – If true, the keys of returned QUBO are indexed with a positive integer number.
- **feed_dict** (*dict[str, float]*) – If the expression contains *Placeholder* objects, you have to specify the value of them by *Placeholder*.

Returns dimod.BinaryQuadraticModel with vartype set to *dimod.BINARY*.

Return type dimod.BinaryQuadraticModel

Examples:

```
>>> from pyqubo import Binary, Constraint
>>> from dimod import ExactSolver
>>> a, b = Binary('a'), Binary('b')
>>> H = Constraint(2*a-3*b, "const1") + Constraint(a+b-1, "const2")
>>> model = H.compile()
>>> bqm = model.to_bqm()
>>> sampleset = ExactSolver().sample(bqm)
>>> decoded_samples = model.decode_sampleset(sampleset)
>>> best_sample = min(decoded_samples, key=lambda s: s.energy)
>>> print(best_sample.energy)
-3.0
>>> pprint(best_sample.sample)
{'a': 0, 'b': 1}
>>> pprint(best_sample.constraints())
{'const1': (False, -3.0), 'const2': (True, 0.0)}
```

energy (*solution*, *vartype*, *feed_dict=None*)

Returns energy of the sample.

Parameters

- **sample** (*list[int]/dict[str, int]*) – The sample returned from solvers.
- **vartype** (*str*) – Variable type of the solution. Specify either 'BINARY' or 'SPIN'.
- **feed_dict** (*dict[str, float]*) – Specify the placeholder values.

Returns Calculated energy.

Return type float

decode_sample (*sample*, *vartype*, *feed_dict=None*)

Decode sample from solvers.

Parameters

- **sample** (*list[int]/dict[str, int]*) – The sample returned from solvers.
- **vartype** (*str*) – Variable type of the solution. Specify either 'BINARY' or 'SPIN'.
- **feed_dict** (*dict[str, float]*) – Specify the placeholder values.

Returns *DecodedSample* object.

Return type *DecodedSample*

Examples

```
>>> from pyqubo import Binary, SubH
>>> a, b = Binary('a'), Binary('b')
>>> H = SubH(a+b-2, "subh1") + 2*a + b
>>> model = H.compile()
```

(continues on next page)

(continued from previous page)

```
>>> decoded_sample = model.decode_sample({'a': 1, 'b': 0}, vartype='BINARY')
>>> print(decoded_sample.energy)
1.0
>>> pprint(decoded_sample.sample)
{'a': 1, 'b': 0}
>>> print(decoded_sample.subh)
{'subh1': -1.0}
```

decode_sampleset (*sampleset*, *feed_dict=None*)

Decode the sample represented by `dimod.SampleSet`.

For more details about `dimod.SampleSet`, see dimod.SampleSet.

Parameters

- **sample** (*dimod.SampleSet*) – The solution returned from dimod sampler.
- **feed_dict** (*dict[str,float]*) – Specify the placeholder values. Default=None

Returns *DecodedSample* object.

Return type *DecodedSample*

Examples

```
>>> from pyqubo import Binary, Constraint
>>> from dimod import ExactSolver
>>> a, b = Binary('a'), Binary('b')
>>> H = Constraint(2*a-3*b, "const1") + Constraint(a+b-1, "const2")
>>> model = H.compile()
>>> bqm = model.to_bqm()
>>> sampleset = ExactSolver().sample(bqm)
>>> decoded_samples = model.decode_sampleset(sampleset)
>>> best_sample = min(decoded_samples, key=lambda s: s.energy)
>>> print(best_sample.energy)
-3.0
>>> pprint(best_sample.sample)
{'a': 0, 'b': 1}
>>> pprint(best_sample.constraints())
{'const1': (False, -3.0), 'const2': (True, 0.0)}
```

5.4.2 DecodedSample

class DecodedSample

DecodedSample contains the informatin like whether the constraint is satisfied or not, or the value of the Sub-Hamiltonian.

Examples

```
>>> from pyqubo import Binary, SubH
>>> a, b = Binary('a'), Binary('b')
>>> H = SubH(a+b-2, "subh1") + 2*a + b
>>> model = H.compile()
>>> decoded_sample = model.decode_sample({'a': 1, 'b': 0}, vartype='BINARY')
>>> print(decoded_sample.energy)
1.0
>>> pprint(decoded_sample.sample)
```

(continues on next page)

(continued from previous page)

```
{'a': 1, 'b': 0}
>>> print(decoded_sample.subh)
{'subh1': -1.0}
```

Methods

array()	Get the value of the array element specified.
constraints()	Returns the information about constraints.

array (*array_name*, *index*)

Get the value of the array specified by *array_name* and *index*.

Parameters

- **array_name** (*str*) – The name of the array.
- **index** (*int/tuple*) – The index of the array.

Returns The value of the array calculated by the sample.

Return type float

Examples

```
>>> from pyqubo import Array
>>> x = Array.create('x', shape=(2, 1), vartype="BINARY")
>>> H = (x[0, 0] + x[1, 0] - 1)**2
>>> model = H.compile()
>>> qubo, offset = model.to_qubo()
>>> pprint(qubo)
{('x[0][0]', 'x[0][0]'): -1.0,
('x[0][0]', 'x[1][0]'): 2.0,
('x[1][0]', 'x[1][0]'): -1.0}
>>> dec = model.decode_sample({'x[0][0]': 1, 'x[1][0]': 0}, vartype='BINARY')
>>> print(dec.array('x', (0, 0)))
1
>>> print(dec.array('x', (1, 0)))
0
```

constraints (*only_broken*)

Get the value of the array specified by *array_name* and *index*.

Parameters **only_broken** (*bool*) – Whether to select only broken constraints.

Returns Dictionary with the key being the label of the constraint and the value being the boolean and the corresponding energy value. The boolean value indicates whether the constraint is satisfied or not.

Return type dict[str, tuple[bool, float]]

Examples

```
>>> from pyqubo import Binary, Constraint
>>> a, b = Binary('a'), Binary('b')
>>> H = Constraint(a+b-2, "const1") + Constraint(a+b-1, "const2")
>>> model = H.compile()
>>> dec = model.decode_sample({'a': 1, 'b': 0}, vartype='BINARY')
>>> pprint(dec.constraints())
{'const1': (False, -1.0), 'const2': (True, 0.0)}
```

(continues on next page)

(continued from previous page)

```
>>> pprint(dec.constraints(only_broken=True))
{'const1': (False, -1.0)}
```

5.5 Array

class Array(*bit_list*)

Multi-dimensional array.

Parameters **bit_list** (list/`numpy.ndarray`) – The object from which a new array is created. Accepted input:

- (Nested) list of `Express`, *`Array`*, int or float.
- numpy.ndarray

shape

Shape of this array.

Type tuple[int]

Example

Create a new array with Binary.

```
>>> from pyqubo import Array, Binary
>>> Array.create('x', shape=(2, 2), vartype='BINARY')
Array([[Binary(x[0][0]), Binary(x[0][1])],
       [Binary(x[1][0]), Binary(x[1][1])]])
```

Create a new array from a nested list of `Express`.

```
>>> array = Array([[Binary('x0'), Binary('x1')], [Binary('x2'), Binary('x3')]])
>>> array
Array([[Binary(x0), Binary(x1)],
       [Binary(x2), Binary(x3)]])
```

Get the shape of the array.

```
>>> array.shape
(2, 2)
```

Access an element with index.

```
>>> array[0, 0]  # = array[(0, 0)]
Binary(x0)
```

Use slice ":" to select a subset of the array.

```
>>> array[:, 1]  # = array[(slice(None), 1)]
Array([Binary(x1), Binary(x3)])
>>> sum(array[:, 1])
(Binary(x1)+Binary(x3))
```

Use list or tuple to select a subset of the array.

```
>>> array[[0, 1], 1]
Array([Binary(x1), Binary(x3)])
>>> array[(0, 1), 1]
Array([Binary(x1), Binary(x3)])
```

Create an array from numpy array.

```
>>> import numpy as np
>>> Array(np.array([[1, 2], [3, 4]]))
Array([[1, 2],
       [3, 4]])
```

Create an array from list of *Array*.

```
>>> Array([Array([1, 2]), Array([3, 4])])
Array([[1, 2],
       [3, 4]])
```

static Array.**create**(*name*, *shape*, *vartype*)

Create a new array with Spins or Binary.

Parameters

- **name** (*str*) – Name of the matrix. It is used as a part of the label of variables. For example, if the name is 'x', the label of *(i, j)* th variable will be `x[i][j]`.
- **shape** (*int/tuple[int]*) – Dimensions of the array.
- **vartype** (dimod.Vartype/str/set, optional) – Variable type of the solution. Accepted input values:
  - `Vartype.SPIN`, `'SPIN'`, `{-1, 1}`
  - `Vartype.BINARY`, `'BINARY'`, `{0, 1}`

Example

```
>>> from pyqubo import Array
>>> array = Array.create('x', shape=(2, 2), vartype='BINARY')
>>> array # doctest: +SKIP
Array([[Binary(x[0][0]), Binary(x[0][1])],
       [Binary(x[1][0]), Binary(x[1][1])]])
>>> array[0] # doctest: +SKIP
Array([Binary(x[0][0]), Binary(x[0][1])])
```

5.5.1 Matrix Operation

Array.T	Returns a transposed array.
Array.dot(other)	Returns a dot product of two arrays.
Array.matmul(other)	Returns a matrix product of two arrays.
Array.reshape(new_shape)	Returns a reshaped array.

pyqubo.Array.T

pyqubo.Array.dot

Array.**dot**(*other*)

Returns a dot product of two arrays.

Parameters **other** (*Array*) – Array.

Returns Express/*Array*

Example

Dot calculation falls into four patterns.

1. If both *self* and *other* are 1-D arrays, it is inner product of vectors.

```
>>> from pyqubo import Array, Binary
>>> array_a = Array([Binary('a'), Binary('b')])
>>> array_b = Array([Binary('c'), Binary('d')])
>>> array_a.dot(array_b) # doctest: +SKIP
((Binary(a)*Binary(c))+(Binary(b)*Binary(d)))
```

2. If *self* is an N-D array and *other* is a 1-D array, it is a sum product over the last axis of *self* and *other*.

```
>>> array_a = Array([[Binary('a'), Binary('b')], [Binary('c'), Binary('d')]])
>>> array_b = Array([Binary('e'), Binary('f')])
>>> array_a.dot(array_b) # doctest: +SKIP
Array([((Binary(a)*Binary(e))+(Binary(b)*Binary(f))),
 ↪((Binary(c)*Binary(e))+(Binary(d)*Binary(f)))])
```

3. If both *self* and *other* are 2-D arrays, it is matrix multiplication.

```
>>> array_a = Array([[Binary('a'), Binary('b')], [Binary('c'), Binary('d')]])
>>> array_b = Array([[Binary('e'), Binary('f')], [Binary('g'), Binary('h')]])
>>> array_a.dot(array_b) # doctest: +SKIP
Array([[((Binary(a)*Binary(e))+(Binary(b)*Binary(g))),
 ↪((Binary(a)*Binary(f))+(Binary(b)*Binary(h)))],
       [((Binary(c)*Binary(e))+(Binary(d)*Binary(g))),
 ↪((Binary(c)*Binary(f))+(Binary(d)*Binary(h)))]])
```

4. If *self* is an N-D array and *other* is an M-D array (where N, M>=2), it is a sum product over the last axis of *self* and the second-to-last axis of *other*. If N = M = 3, (i, j, k, m) element of a dot product of *self* and *other* is:

```
dot(self, other)[i,j,k,m] = sum(self[i,j,:] * other[k,:,m])
```

```
>>> array_a = Array.create('a', shape=(3, 2, 4), vartype='BINARY')
>>> array_a.shape
(3, 2, 4)
>>> array_b = Array.create('b', shape=(5, 4, 3), vartype='BINARY')
>>> array_b.shape
(5, 4, 3)
>>> i, j, k, m = (1, 1, 3, 2)
>>> array_a.dot(array_b)[i, j, k, m] == sum(array_a[i, j, :] * array_b[k, :, m])
True
```

Dot product with list.

```
>>> array_a = Array([Binary('a'), Binary('b')])
>>> array_b = [3, 4]
>>> array_a.dot(array_b) # doctest: +SKIP
((Binary(a)*Num(3))+(Binary(b)*Num(4)))
```

pyqubo.Array.matmul

Array.**matmul**(*other*)

Returns a matrix product of two arrays.

Note: You can use operator symbol '@' instead of *matmul()* in Python 3.5 or later version.

```
>>> from pyqubo import Array
>>> array_a = Array.create('a', shape=(2, 4), vartype='BINARY')
>>> array_b = Array.create('b', shape=(4, 3), vartype='BINARY')
>>> array_a @ array_b == array_a.matmul(array_b)
True
```

Parameters **other** (*Array*/numpy.ndarray/list) –

Returns *Array*/Express

Example

Matrix product of two arrays falls into 3 patterns.

1. If either of the arguments is 1-D array, it is treated as a matrix where one is added to its dimension.

```
>>> from pyqubo import Array, Binary
>>> array_a = Array([[Binary('a'), Binary('b')], [Binary('c'), Binary('d')]])
>>> array_b = Array([Binary('e'), Binary('f')])
>>> array_a.matmul(array_b) # doctest: +SKIP
Array([((Binary(a)*Binary(e))+(Binary(b)*Binary(f))),
 ↪((Binary(c)*Binary(e))+(Binary(d)*Binary(f)))])
```

2. If both arguments are 2-D array, conventional matrix product is calculated.

```
>>> array_a = Array([[Binary('a'), Binary('b')], [Binary('c'), Binary('d')]])
>>> array_b = Array([[Binary('e'), Binary('f')], [Binary('g'), Binary('h')]])
>>> array_a.matmul(array_b) # doctest: +SKIP
Array([[((Binary(a)*Binary(e))+(Binary(b)*Binary(g))),
 ↪((Binary(a)*Binary(f))+(Binary(b)*Binary(h)))],
       [((Binary(c)*Binary(e))+(Binary(d)*Binary(g))),
 ↪((Binary(c)*Binary(f))+(Binary(d)*Binary(h)))]])
```

3. If either argument is N-D (where N > 2), it is treated as an array whose element is a 2-D matrix of last two indices. In this example, *array_a* is treated as if it is a vector whose elements are two matrices of shape (2, 3).

```
>>> array_a = Array.create('a', shape=(2, 2, 3), vartype='BINARY')
>>> array_b = Array.create('b', shape=(3, 2), vartype='BINARY')
>>> (array_a @ array_b)[0] == array_a[0].matmul(array_b)
True
```

pyqubo.Array.reshape

Array.**reshape**(*new_shape*)

Returns a reshaped array.

Parameters **new_shape** (*tuple[int]*) – New shape.

Example

```
>>> from pyqubo import Array
>>> array = Array.create('x', shape=(2, 3), vartype='BINARY')
>>> array
Array([[Binary(x[0][0]), Binary(x[0][1]), Binary(x[0][2])],
       [Binary(x[1][0]), Binary(x[1][1]), Binary(x[1][2])]])
>>> array.reshape((3, 2, 1))
Array([[[Binary(x[0][0])],
        [Binary(x[0][1])]],
       [[Binary(x[0][2])],
        [Binary(x[1][0])]],
       [[Binary(x[1][1])],
        [Binary(x[1][2])]]])
```

5.5.2 Arithmetic Operation

Array.add(other)	Returns a sum of self and other.
Array.subtract(other)	Returns a difference between other and self.
Array.mul(other)	Returns a multiplicity of self by other.
Array.div(other)	Returns division of self by other.

pyqubo.Array.add

Array.**add**(*other*)

Returns a sum of self and other.

Parameters **other** (*Array*/ndarray/int/float) – Addend.

Returns *Array*

Example

```
>>> from pyqubo import Array, Binary
>>> import numpy as np
>>> array_a = Array([[Binary('a'), Binary('b')], [Binary('c'), 2]])
>>> array_b = Array([[Binary('d'), 1], [Binary('f'), Binary('g')]])
>>> array_a.add(array_b) # doctest: +SKIP
```

(continues on next page)

(continued from previous page)

```
Array([[(Binary(a)+Binary(d)), (Binary(b)+Num(1))],
       [(Binary(c)+Binary(f)), (Binary(g)+Num(2))]])
>>> array_a + array_b # doctest: +SKIP
Array([[(Binary(a)+Binary(d)), (Binary(b)+Num(1))],
       [(Binary(c)+Binary(f)), (Binary(g)+Num(2))]])
```

Sum of self and scalar value.

```
>>> array_a + 5 # doctest: +SKIP
Array([[(Binary(a)+Num(5)), (Binary(b)+Num(5))],
       [(Binary(c)+Num(5)), 7]])
```

Sum of self and numpy ndarray.

```
>>> array_a + np.array([[1, 2], [3, 4]]) # doctest: +SKIP
Array([[(Binary(a)+Num(1)), (Binary(b)+Num(2))],
       [(Binary(c)+Num(3)), 6]])
```

pyqubo.Array.subtract

Array.**subtract**(*other*)

Returns a difference between other and self.

Parameters **other** (*Array*/ndarray/int/float) – Subtrahend.

Returns *Array*

Example

```
>>> from pyqubo import Array, Binary
>>> import numpy as np
>>> array_a = Array([[Binary('a'), Binary('b')], [Binary('c'), 2]])
>>> array_b = Array([[Binary('d'), 1], [Binary('f'), Binary('g')]])
>>> array_a.subtract(array_b) # doctest: +SKIP
Array([[(Binary(a)+(Binary(d)*Num(-1))), (Binary(b)+Num(-1))],
       [(Binary(c)+(Binary(f)*Num(-1))), ((Binary(g)*Num(-1))+Num(2))]])
>>> array_a - array_b # doctest: +SKIP
Array([[(Binary(a)+(Binary(d)*Num(-1))), (Binary(b)+Num(-1))],
       [(Binary(c)+(Binary(f)*Num(-1))), ((Binary(g)*Num(-1))+Num(2))]])
```

Difference of self and scalar value.

```
>>> array_a - 5 # doctest: +SKIP
Array([[(Binary(a)+Num(-5)), (Binary(b)+Num(-5))],
       [(Binary(c)+Num(-5)), -3]])
```

Difference of self and numpy ndarray.

```
>>> array_a - np.array([[1, 2], [3, 4]]) # doctest: +SKIP
Array([[(Binary(a)+Num(-1)), (Binary(b)+Num(-2))],
       [(Binary(c)+Num(-3)), -2]])
```

pyqubo.Array.mul

Array.**mul**(*other*)

Returns a multiplicity of self by other.

Parameters **other** (*Array*/ndarray/int/float) – Factor.

Returns *Array*

Example

```
>>> from pyqubo import Array, Binary
>>> import numpy as np
>>> array_a = Array([[Binary('a'), Binary('b')], [Binary('c'), 2]])
>>> array_b = Array([[Binary('d'), 1], [Binary('f'), Binary('g')]])
>>> array_a.mul(array_b) # doctest: +SKIP
Array([[(Binary(a)*Binary(d)), (Binary(b)*Num(1))],
       [(Binary(c)*Binary(f)), (Binary(g)*Num(2))]])
>>> array_a * array_b # doctest: +SKIP
Array([[(Binary(a)*Binary(d)), (Binary(b)*Num(1))],
       [(Binary(c)*Binary(f)), (Binary(g)*Num(2))]])
```

Product of self and scalar value.

```
>>> array_a * 5 # doctest: +SKIP
Array([[(Binary(a)*Num(5)), (Binary(b)*Num(5))],
       [(Binary(c)*Num(5)), 10]])
```

Product of self and numpy ndarray.

```
>>> array_a * np.array([[1, 2], [3, 4]]) # doctest: +SKIP
Array([[(Binary(a)*Num(1)), (Binary(b)*Num(2))],
       [(Binary(c)*Num(3)), 8]])
```

pyqubo.Array.div

Array.**div**(*other*)

Returns division of self by other.

Parameters **other** (*int/float*) – Divisor.

Returns *Array*

Example

```
>>> from pyqubo import Array, Binary
>>> array_a = Array([[Binary('a'), Binary('b')], [Binary('c'), 2]])
>>> array_a / 5 # doctest: +SKIP
Array([[(Binary(a)*Num(0.2)), (Binary(b)*Num(0.2))],
       [(Binary(c)*Num(0.2)), 0.4]])
```

5.5.3 Construction

Array.fill(obj, shape)	Create a new array with the given shape, all filled with the given object.

pyqubo.Array.fill

static Array.**fill**(*obj*, *shape*)

Create a new array with the given shape, all filled with the given object.

Parameters

- **obj** (int/float/Express) – The object with which a new array is filled.
- **shape** (*tuple[int]*) – Shape of the array.

Returns Created array.

Return type *Array*

Example

```
>>> from pyqubo import Array, Binary
>>> Array.fill(Binary('a'), shape=(2, 3))
Array([[Binary(a), Binary(a), Binary(a)],
       [Binary(a), Binary(a), Binary(a)]])
```

5.6 Integer

Summary of each integer encoding, whose value takes [0, n].

Encoding	Value	Constraint	#vars	Max abs. coeff of value
UnaryEncInteger	$\sum_{i=1}^{n} x_i$	No constraint	n	1
LogEncInteger	$\sum_{i=1}^{d} 2^i x_i$	No constraint	$\lceil \log_2 n \rceil (= d)$	2^d
OneHotEncInteger	$\sum_{i=0}^{n} i x_i$	$(\sum_{i=0}^{n} x_i - 1)^2$	$n+1$	n
OrderEncInteger	$\sum_{i=1}^{n} x_i$	$\sum_{i=1}^{n-1} x_{i+1}(1-x_i)$	n	1

5.6.1 UnaryEncInteger

class UnaryEncInteger(*label*, *value_range*)

Unary encoded integer. The value that takes $[0, n]$ is represented by $\sum_{i=1}^{n} x_i$ without any constraint.

Parameters

- **label** (*str*) – Label of the integer.
- **lower** (*int*) – Lower value of the integer.
- **upper** (*int*) – Upper value of the integer.

Examples

This example finds the value a, b such that $a + b = 3$ and $2a - b = 0$.

```
>>> from pyqubo import UnaryEncInteger
>>> import dimod
>>> a = UnaryEncInteger("a", (0, 3))
>>> b = UnaryEncInteger("b", (0, 3))
>>> M=2.0
>>> H = (2*a-b)**2 + M*(a+b-3)**2
>>> model = H.compile()
>>> bqm = model.to_bqm()
>>> import dimod
>>> sampleset = dimod.ExactSolver().sample(bqm)
>>> decoded_samples = model.decode_sampleset(sampleset)
>>> best_sample = min(decoded_samples, key=lambda s: s.energy)
>>> print(best_sample.subh['a'])
1.0
>>> print(best_sample.subh['b'])
2.0
```

5.6.2 LogEncInteger

class LogEncInteger(*label*, *value_range*)

Log encoded integer. The value that takes $[0, n]$ is represented by $\sum_{i=1}^{\lceil \log_2 n \rceil} 2^i x_i$ without any constraint.

Parameters

- **label** (*str*) – Label of the integer.
- **lower** (*int*) – Lower value of the integer.
- **upper** (*int*) – Upper value of the integer.

Examples

This example finds the value a, b such that $a + b = 5$ and $2a - b = 1$.

```
>>> from pyqubo import LogEncInteger
>>> import dimod
>>> a = LogEncInteger("a", (0, 4))
>>> b = LogEncInteger("b", (0, 4))
>>> M=2.0
>>> H = (2*a-b-1)**2 + M*(a+b-5)**2
>>> model = H.compile()
>>> bqm = model.to_bqm()
>>> import dimod
>>> sampleset = dimod.ExactSolver().sample(bqm)
>>> decoded_samples = model.decode_sampleset(sampleset)
>>> best_sample = min(decoded_samples, key=lambda s: s.energy)
>>> print(best_sample.subh['a'])
2.0
>>> print(best_sample.subh['b'])
3.0
```

5.6.3 OneHotEncInteger

class OneHotEncInteger(*label*, *value_range*, *strength*)

One-hot encoded integer. The value that takes $[1, n]$ is represented by $\sum_{i=1}^{n} ix_i$. Also we have the penalty function $strength \times (\sum_{i=1}^{n} x_i - 1)^2$ in the Hamiltonian.

Parameters

- **label** (*str*) – Label of the integer.
- **lower** (*int*) – Lower value of the integer.
- **upper** (*int*) – Upper value of the integer.
- **strength** (*float/Placeholder*) – Strength of the constraint.

Examples

This example is equivalent to the following Hamiltonian.

$$H = \left(\left(\sum_{i=1}^{3} ia_i + 1\right) - 2\right)^2 + strength \times \left(\sum_{i=1}^{3} a_i - 1\right)^2$$

```
>>> from pyqubo import OneHotEncInteger
>>> a = OneHotEncInteger("a", (1, 3), strength=5)
>>> H = (a-2)**2
>>> model = H.compile()
>>> bqm = model.to_bqm()
>>> import dimod
>>> sampleset = dimod.ExactSolver().sample(bqm)
>>> decoded_samples = model.decode_sampleset(sampleset)
>>> best_sample = min(decoded_samples, key=lambda s: s.energy)
>>> print(best_sample.subh['a'])
2.0
```

equal_to(*k*)

Variable representing whether the value is equal to *k*.

Note: You cannot use this method alone. You should use this variable with the entire integer.

Parameters **k** (*int*) – Integer value.

Returns Express

5.6.4 OrderEncInteger

class OrderEncInteger(*label*, *value_range*, *strength*)

Order encoded integer. This encoding is useful when you want to know whether the integer is more than k or not. The value that takes $[0, n]$ is represented by $\sum_{i=1}^{n} x_i$. Also we have the penalty function $strength \times \left(\sum_{i=1}^{n-1} (x_{i+1} - x_i x_{i+1})\right)$ in the Hamiltonian. See the reference [TaTK09] for more details.

Parameters

- **label** (*str*) – Label of the integer.

- **lower** (*int*) – Lower value of the integer.
- **upper** (*int*) – Upper value of the integer.
- **strength** (*float/Placeholder*) – Strength of the constraint.

Examples

Create an order encoded integer a that takes [0, 3] with the strength = 5.0. Solution of a represents 2 which is the optimal solution of the Hamiltonian.

```
>>> from pyqubo import OrderEncInteger
>>> a = OrderEncInteger("a", (0, 3), strength = 5.0)
>>> model = ((a-2)**2).compile()
>>> bqm = model.to_bqm()
>>> import dimod
>>> sampleset = dimod.ExactSolver().sample(bqm)
>>> decoded_samples = model.decode_sampleset(sampleset)
>>> best_sample = min(decoded_samples, key=lambda s: s.energy)
>>> print(best_sample.subh['a'])
2.0
```

less_than(*k*)

Binary variable that represents whether the value is less than k.

Note: You cannot use this method alone. You should use this variable with the entire integer. See an example below.

Parameters **k** (*int*) – Integer value.

Returns Express

Examples

This example finds the value of integer a and b such that $a = b$ and $a > 1$ and $b < 3$. The obtained solution is $a = b = 2$.

```
>>> from pyqubo import OrderEncInteger
>>> a = OrderEncInteger("a", (0, 4), strength = 5.0)
>>> b = OrderEncInteger("b", (0, 4), strength = 5.0)
>>> model = ((a-b)**2 + (1-a.more_than(1))**2 + (1-b.less_than(3))**2).
↪compile()
>>> bqm = model.to_bqm()
>>> import dimod
>>> sampleset = dimod.ExactSolver().sample(bqm)
>>> decoded_samples = model.decode_sampleset(sampleset)
>>> best_sample = min(decoded_samples, key=lambda s: s.energy)
>>> print(best_sample.subh['a'])
2.0
>>> print(best_sample.subh['b'])
2.0
```

more_than(*k*)

Binary variable that represents whether the value is more than k.

Note: You cannot use this method alone. You should use this variable with the entire integer. See an example below.

Parameters **k** (*int*) – Integer value.

Returns Express

Examples

This example finds the value of integer a and b such that $a = b$ and $a > 1$ and $b < 3$. The obtained solution is $a = b = 2$.

```
>>> from pyqubo import OrderEncInteger
>>> a = OrderEncInteger("a", (0, 4), strength = 5.0)
>>> b = OrderEncInteger("b", (0, 4), strength = 5.0)
>>> model = ((a-b)**2 + (1-a.more_than(1))**2 + (1-b.less_than(3))**2).
↪compile()
>>> bqm = model.to_bqm()
>>> import dimod
>>> sampleset = dimod.ExactSolver().sample(bqm)
>>> decoded_samples = model.decode_sampleset(sampleset)
>>> best_sample = min(decoded_samples, key=lambda s: s.energy)
>>> print(best_sample.subh['a'])
2.0
>>> print(best_sample.subh['b'])
2.0
```

References

5.7 Logical Constraint

5.7.1 NOT Constraint

class NotConst(*a*, *b*, *label*)

Constraint: Not(a) = b.

Parameters

- **a** (Express) – expression to be binary
- **b** (Express) – expression to be binary
- **label** (*str*) – label to identify the constraint

Examples

In this example, when the binary variables satisfy the constraint, the energy is 0.0. On the other hand, when they break the constraint, the energy is 1.0 > 0.0.

```
>>> from pyqubo import NotConst, Binary
>>> a, b = Binary('a'), Binary('b')
>>> exp = NotConst(a, b, 'not')
>>> model = exp.compile()
>>> model.energy({'a': 1, 'b': 0}, vartype='BINARY')
0.0
>>> model.energy({'a': 1, 'b': 1}, vartype='BINARY')
1.0
```

5.7.2 AND Constraint

class AndConst(*a*, *b*, *c*, *label*)

Constraint: AND(a, b) = c.

Parameters

- **a** (Express) – expression to be binary
- **b** (Express) – expression to be binary
- **c** (Express) – expression to be binary
- **label** (*str*) – label to identify the constraint

Examples

In this example, when the binary variables satisfy the constraint, the energy is 0.0. On the other hand, when they break the constraint, the energy is $1.0 > 0.0$.

```
>>> from pyqubo import AndConst, Binary
>>> a, b, c = Binary('a'), Binary('b'), Binary('c')
>>> exp = AndConst(a, b, c, 'and')
>>> model = exp.compile()
>>> model.energy({'a': 1, 'b': 0, 'c': 0}, vartype='BINARY')
0.0
>>> model.energy({'a': 0, 'b': 1, 'c': 1}, vartype='BINARY')
1.0
```

5.7.3 OR Constraint

class OrConst(*a*, *b*, *c*, *label*)

Constraint: OR(a, b) = c.

Parameters

- **a** (Express) – expression to be binary
- **b** (Express) – expression to be binary
- **c** (Express) – expression to be binary
- **label** (*str*) – label to identify the constraint

Examples

In this example, when the binary variables satisfy the constraint, the energy is 0.0. On the other hand, when they break the constraint, the energy is 1.0 > 0.0.

```
>>> from pyqubo import OrConst, Binary
>>> a, b, c = Binary('a'), Binary('b'), Binary('c')
>>> exp = OrConst(a, b, c, 'or')
>>> model = exp.compile()
>>> model.energy({'a': 1, 'b': 0, 'c': 1}, vartype='BINARY')
0.0
>>> model.energy({'a': 0, 'b': 1, 'c': 0}, vartype='BINARY')
1.0
```

5.7.4 XOR Constraint

class XorConst(*a*, *b*, *c*, *label*)

Constraint: OR(a, b) = c.

Parameters

- **a** (Express) – expression to be binary
- **b** (Express) – expression to be binary
- **c** (Express) – expression to be binary
- **label** (*str*) – label to identify the constraint

Examples

In this example, when the binary variables satisfy the constraint, the energy is 0.0. On the other hand, when they break the constraint, the energy is 1.0 > 0.0.

```
>>> from pyqubo import XorConst, Binary
>>> a, b, c = Binary('a'), Binary('b'), Binary('c')
>>> exp = XorConst(a, b, c, 'xor')
>>> model = exp.compile()
>>> model.energy({'a': 1, 'b': 0, 'c': 1, 'aux_xor': 0}, vartype='BINARY')
0.0
>>> model.energy({'a': 0, 'b': 1, 'c': 0, 'aux_xor': 0}, vartype='BINARY')
1.0
```

5.8 Logical Gate

5.8.1 Not

class Not(*bit*)

Logical NOT of input.

Parameters **bit** (Express) – expression to be binary

Examples

```
>>> from pyqubo import Binary, Not
>>> a = Binary('a')
>>> exp = Not(a)
>>> model = exp.compile()
>>> for a in (0, 1):
...   print(a, int(model.energy({'a': a}, vartype='BINARY')))
0 1
1 0
```

5.8.2 And

class And(*bit_a*, *bit_b*)

Logical AND of inputs.

Parameters

- **bit_a** (Express) – expression to be binary
- **bit_b** (Express) – expression to be binary

Examples

```
>>> from pyqubo import Binary, And
>>> import itertools
>>> a, b = Binary('a'), Binary('b')
>>> exp = And(a, b)
>>> model = exp.compile()
>>> for a, b in itertools.product(*[(0, 1)] * 2):
...   print(a, b, int(model.energy({'a': a, 'b': b}, vartype='BINARY')))
0 0 0
0 1 0
1 0 0
1 1 1
```

5.8.3 Or

class Or(*bit_a*, *bit_b*)

Logical OR of inputs.

Parameters

- **bit_a** (Express) – expression to be binary
- **bit_b** (Express) – expression to be binary

Examples

```
>>> from pyqubo import Binary, Or
>>> import itertools
>>> a, b = Binary('a'), Binary('b')
>>> exp = Or(a, b)
```

(continues on next page)

(continued from previous page)

```
>>> model = exp.compile()
>>> for a, b in itertools.product(*[(0, 1)] * 2):
...   print(a, b, int(model.energy({'a': a, 'b': b}, vartype='BINARY')))
0 0 0
0 1 1
1 0 1
1 1 1
```

5.8.4 Xor

class Xor(*bit_a*, *bit_b*)

Logical XOR of inputs.

Parameters

- **bit_a** (Express) – expression to be binary
- **bit_b** (Express) – expression to be binary

Examples

```
>>> from pyqubo import Binary, Xor
>>> import itertools
>>> a, b = Binary('a'), Binary('b')
>>> exp = Xor(a, b)
>>> model = exp.compile()
>>> for a, b in itertools.product(*[(0, 1)] * 2):
...   print(a, b, int(model.energy({'a': a, 'b': b}, vartype='BINARY')))
0 0 0
0 1 1
1 0 1
1 1 0
```

5.9 Utils

5.9.1 Solvers

solve_ising(*linear*, *quad*, *num_reads=10*, *sweeps=1000*, *beta_range=(1.0, 50.0)*)

[deprecated] Solve Ising model with Simulated Annealing (SA) provided by neal.

Parameters

- **linear** (*dict[label, float]*) – The linear parameter of the Ising model.
- **quad** (*dict[(label, label), float]*) – The quadratic parameter of the Ising model.
- **num_reads** (*int, default=10*) – Number of run repetitions of SA.
- **sweeps** (*int, default=1000*) – Number of iterations in each run of SA.
- **beta_range** (*tuple(float, float), default=(1.0, 50.0)*) – Tuple of start beta and end beta.

Note: *solve_ising()* is deprecated. Use *dwave-neal* package instead like below.

```
>>> from pyqubo import Spin
>>> import neal
>>> s1, s2, s3 = Spin("s1"), Spin("s2"), Spin("s3")
>>> H = (2*s1 + 4*s2 + 6*s3)**2
>>> model = H.compile()
>>> bqm = model.to_bqm()
>>> sa = neal.SimulatedAnnealingSampler()
>>> sampleset = sa.sample(bqm, num_reads=10)
>>> samples = model.decode_sampleset(sampleset)
>>> best_sample = min(samples, key=lambda s: s.energy)
>>> pprint(best_sample.sample) # doctest: +SKIP
{'s1': 0, 's2': 0, 's3': 1}
```

solve_qubo(*qubo*, *num_reads=10*, *sweeps=1000*, *beta_range=(1.0, 50.0)*)
[deprecated] Solve QUBO with Simulated Annealing (SA) provided by neal.

Parameters

- **qubo** (*dict[(label, label), float]*) – The QUBO to be solved.
- **num_reads** (*int, default=10*) – Number of run repetitions of SA.
- **sweeps** (*int, default=1000*) – Number of iterations in each run of SA.
- **beta_range** (*tuple(float, float), default=(1.0, 50.0)*) – Tuple of start beta and end beta.

Returns The solution of SA.

Return type dict[label, bit]

Note: *solve_qubo()* is deprecated. Use *dwave-neal* package instead like below.

```
>>> from pyqubo import Spin
>>> import neal
>>> s1, s2, s3 = Spin("s1"), Spin("s2"), Spin("s3")
>>> H = (2*s1 + 4*s2 + 6*s3)**2
>>> model = H.compile()
>>> bqm = model.to_bqm()
>>> sa = neal.SimulatedAnnealingSampler()
>>> sampleset = sa.sample(bqm, num_reads=10)
>>> samples = model.decode_sampleset(sampleset)
>>> best_sample = min(samples, key=lambda s: s.energy)
>>> pprint(best_sample.sample) # doctest: +SKIP
{'s1': 0, 's2': 0, 's3': 1}
```

5.9.2 Asserts

assert_qubo_equal(*qubo1*, *qubo2*)
Assert the given QUBOs are identical.

Parameters

- **qubo1** (*dict[(label, label), float]*) – QUBO to be compared.

- **qubo2** (*dict[(label, label), float]*) – QUBO to be compared.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Bibliography

[TaTK09] Tamura, N., Taga, A., Kitagawa, S., & Banbara, M. (2009). Compiling finite linear CSP into SAT. Constraints, 14(2), 254-272.

Python Module Index

p

Index

U

W

X