pyqubo Documentation

Release 0.0.1

Author

Apr 01, 2019

Manual:

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

- **Python-based**.
- **QUBO generation (compile) is fast.**
- **Automatic validation of constraints.** (details)
- **Placeholder** for parameter tuning. (details)

For more details, see PyQUBO Documentation.

CHAPTER 1

Example Usage

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,
 ('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}
```

For more examples, see example notebooks.

CHAPTER 2

Installation

```
pip install pyqubo
```

or

```
python setup.py install
```

CHAPTER 3

Supported Python Versions

Python 2.7, 3.4, 3.5, 3.6 and 3.7 are supported.

3.1 Getting Started

3.1.1 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
```

3.1.2 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 $q_{ii} x_i x_i = q_{ii} x_i$, 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.

3.1.3 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 `Model.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.

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

3.1.4 Variable: Binary and Spin

When you define a hamiltonian, you can use *Binary* or *Spin* as a 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')
>>> exp = 2
```


```
>>> from pyqubo import Binary
>>> x1, x2 = Binary('x1'), Binary('x2')
>>> exp = 2
```


```
>>> from pyqubo import Binary
>>> x1, x2 = Binary('x1'), Binary('x2')
>>> exp = 2*x1*x2 + 3*x1
>>> pprint(exp.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')
>>> exp = 2*s1*s2 + 3*s1
>>> pprint(exp.compile().to_qubo()) # doctest: +SKIP
({('s1', 's1'): 2.0, ('s1', 's2'): 8.0, ('s2', 's2'): -4.0}, -1.0)
```

3.1.5 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}
```

3.1.6 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
```

```
>>> from pyqubo import Binary
>>> a, b = Binary('a'), Binary('b')
>>> M = 5.0
>>> H = 2
```

```
>>> 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()
```

(continues on next page)

(continued from previous page)

```
>>> 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.

3.1.7 Decode Solution

When you get a solution from the Ising solver, *Model.decode_solution()* decodes the solution and returns decoded_solution in dictionary form.

Example: You are solving a partitioning problem.

```
>>> from pyqubo import Array
>>> numbers = [4, 2, 7, 1]
>>> s = Array.create('s', 4, 'SPIN')
>>> H = sum(n * s_i for s_i, n in zip(s, numbers))**2
>>> model = H.compile()
>>> qubo, offset = model.to_qubo()
```

Let's assume that you get a solution `{'s[0]': 0, 's[1]': 0, 's[2]': 1, 's[3]': 0}` from the solver.

```
>>> raw_solution = {'s[0]': 0, 's[1]': 0, 's[2]': 1, 's[3]': 0} # solution from the␣
↪solver
>>> decoded_solution, broken, energy = model.decode_solution(raw_solution, vartype=
↪'BINARY')
>>> pprint(decoded_solution)
{'s': {0: 0, 1: 0, 2: 1, 3: 0}}
>>> broken
{}
>>> energy
0.0
```

You can see that `decoded_solution` has the decoded solution of spin vector where i th element of the vector is accessed via *s[i]*. `broken` represents broken constraint which will be explained in the following section. `energy` represents energy of the problem.

3.1.8 Validation of Constraints

When the hamiltonian has constraints, you can let the compiler recognize the hamiltonian of the constraint with `Constraint`. When you decode the solution, the model let you know which constraints are broken. You don't have to write additional programs for validation of the constraints.

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 put $(a + b - 1)^2$ in `Constraint` to tell the compiler that this hamiltonian is constraint i.e. it should be zero when the solution is not broken.

```
>>> 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

```
>>> 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

```
>>> 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': 1, 'b': 1}` from the solver which breaks the constraint $a + b = 1$.

```
>>> raw_solution = {'a': 1, 'b': 1}
>>> decoded_solution, broken, energy = model.decode_solution(raw_solution, vartype=
↪'BINARY')
>>> pprint(broken)
{'a+b=1': {'penalty': 1.0, 'result': {'a': 1, 'b': 1}}}
```

`broken` object contains the information about the broken constraint. If no constraint is broken, `broken` is empty.

3.2 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.

3.2.1 Installation

If you already installed PyQUBO, uninstall it.

```
pip uninstall pyqubo
```

Install PyQUBO with development mode

```
python setup.py develop
```

3.2.2 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.

3.2.3 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 test
```

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/.

3.2.4 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

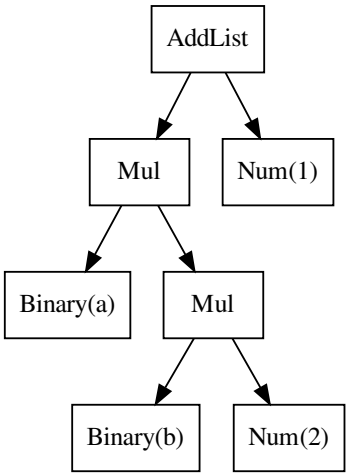
```
make doctest
```

3.3 Expression

class Express

Abstract class of pyqubo expression.

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



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 *Express*.

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

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 *Express*.

Return type *Model*

Examples

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

3.3.1 Binary

class Binary(*label*, *structure=None*)

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

Parameters

- **label** (*str*) – The label of a variable. A variable is identified by this label.
- **structure** (*dict/optional*) – Variable structure.

Example

```
>>> 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)
```

3.3.2 Spin

class Spin(*label*, *structure=None*)

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

Parameters

- **label** (*str*) – The label of a variable. A variable is identified by this label.
- **structure** (*dict/optional*) – Variable structure.

Example

```
>>> 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)
```

3.3.3 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)
```

3.3.4 Constraint

class Constraint(*child*, *label*)

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.

Example

When the solution is broken, *decode_solution* can detect it. In this example, we introduce a constraint $a+b=1$.

```
>>> from pyqubo import Binary, Constraint
>>> a, b = Binary('a'), Binary('b')
>>> exp = a + b + Constraint((a+b-1)**2, label="one_hot")
>>> model = exp.compile()
>>> sol, broken, energy = model.decode_solution({'a': 1, 'b': 1}, vartype='BINARY
↪')
>>> pprint(broken)
{'one_hot': {'penalty': 1.0, 'result': {'a': 1, 'b': 1}}}
>>> sol, broken, energy = model.decode_solution({'a': 1, 'b': 0}, vartype='BINARY
↪')
>>> pprint(broken)
{}
```

3.3.5 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
>>> class LogicalAnd(UserDefinedExpress):
...     def __init__(self, bit_a, bit_b):
...         self._express = bit_a * bit_b
...
...     @property
...     def express(self):
...         return self._express
```

express

Expression of the Hamiltonian defined by the user.

Type *Express*

3.3.6 Add

class Add(*left*, *right*)

Addition of expressions (deprecated).

Parameters

- **left** (*Express*) – An expression
- **right** (*Express*) – 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))
```

3.3.7 AddList

class AddList(*terms*)

Addition of a list of expressions.

Parameters **terms** (list[*Express*]) – a list of expressions

Example

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

```
>>> from pyqubo import Binary, AddList
>>> a, b = Binary('a'), Binary('b')
>>> a + b
(Binary(a)+Binary(b))
>>> AddList([a, b])
(Binary(a)+Binary(b))
```

3.3.8 Mul

class Mul(*left*, *right*)

Product of expressions.

Parameters

- **left** (*Express*) – An expression
- **right** (*Express*) – 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))
```

3.3.9 Num

class Num(*value*)

Expression of number

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

Example

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

3.4 Model

class Model(*compiled_qubo*, *structure*, *constraints*)

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.

Parameters

- **compiled_qubo** (*CompiledQubo*) – Compiled QUBO. If we want to get the final QUBO, we need to evaluate this QUBO by specifying the value of placeholders. See `CompiledQubo.eval()`.
- **structure** (*dict[label, Tuple(key1, key2, key3, ...)]*) – It defines the mapping of the variable used in *decode_solution()*. A solution of *label* is mapped to `decoded_solution[key1][key2][key3][...]`. For more details, see *decode_solution()*.
- **constraints** (*dict[label, polynomial_term]*) – It contains constraints of the problem. *label* is each constraint name and *polynomial_term* is corresponding polynomial which should be zero when the constraint is satisfied.

variable_order

The list of labels. The order is corresponds to the index of QUBO or Ising model.

Type list

index2label

The dictionary which maps an index to a label.

Type dict[int, label]

label2index

The dictionary which maps a label to an index.

Type dict[label, index]

decode_dimod_response(*response*, *topk=None*, *feed_dict=None*)

Decode the solution of `dimod.Response`.

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

Parameters

- **response** (`dimod.Response`) – The solution returned from dimod sampler.

- **topk** (*int, default=None*) – Decode only top-k (energy is smaller) solutions.
- **feed_dict** (*dict[str, float]*) – Specify the placeholder values. Default=None

Returns List of tuple of the decoded solution and broken constraints and energy. Solutions are sorted by energy. Structure of decoded_solution is defined by `structure`.

Return type list[tuple(dict, dict, float)]

decode_solution(***kwargs*)
Returns decoded solution.

Parameters

- **solution** (*list[bit]/dict[label, bit]/dict[index, bit]*) – The solution returned from solvers.
- **vartype** (`dimod.Vartype`/str/set, optional) – Variable type of the solution. Accepted input values:
  - `Vartype.SPIN`, `'SPIN'`, `{-1, 1}`
  - `Vartype.BINARY`, `'BINARY'`, `{0, 1}`
- **feed_dict** (*dict[str, float]*) – Specify the placeholder values.

Returns Tuple of the decoded solution, broken constraints and energy. Structure of decoded_solution is defined by `structure`.

Return type tuple(dict, dict, float)

energy(***kwargs*)
Returns energy of the solution.

Parameters

- **solution** (*list[bit]/dict[label, bit]/dict[index, bit]*) – The solution returned from solvers.
- **vartype** (`dimod.Vartype`/str/set, optional) – Variable type of the solution. Accepted input values:
  - `Vartype.SPIN`, `'SPIN'`, `{-1, 1}`
  - `Vartype.BINARY`, `'BINARY'`, `{0, 1}`
- **feed_dict** (*dict[str, float]*) – Specify the placeholder values.

Returns energy of the solution.

Return type float

to_dimod_bqm(*feed_dict=None*)
Returns `dimod.BinaryQuadraticModel`.

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

Parameters **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*.

to_ising(*index_label=False*, *feed_dict=None*)
Returns Ising Model and energy offset.

Parameters

- **index_label** (*bool*) – If true, the keys of returned Ising model 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 Tuple of Ising Model and energy offset. Where *linear* takes the form of `(dict[label, value])`, and *quadratic* takes the form of `dict[(label, label), value]`.

Return type tuple(linear, quadratic, float)

Examples

This example creates the `model` from the expression, and we get the resulting Ising model by calling `model.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, ('y', 'z'): 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: 0.25, 1: 0.5, 2: 1.75}, {(0, 1): 0.25, (1, 2): 0.25}, 2.0)
>>> model.variable_order
['x', 'y', 'z']
```

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*.

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

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

(continues on next page)

(continued from previous page)

```
  ('y', 'y'): 0.0,
  ('y', 'z'): 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.variable_order`.

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

3.5 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**(**args*, ***kwargs*)

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

3.5.1 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

Array.**T**

Returns a transposed array.

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.T
Array([[Binary(x[0][0]), Binary(x[1][0])],
       [Binary(x[0][1]), Binary(x[1][1])],
       [Binary(x[0][2]), Binary(x[1][2])]])
```

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)
((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)
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)
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)
((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)
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)
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])]]])
```

3.5.2 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)
Array([[(Binary(a)+Binary(d)), (Binary(b)+Num(1))],
       [(Binary(c)+Binary(f)), (Binary(g)+Num(2))]])
>>> array_a + array_b
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
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]])
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)
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
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
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]])
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)
Array([[(Binary(a)*Binary(d)), (Binary(b)*Num(1))],
       [(Binary(c)*Binary(f)), (Binary(g)*Num(2))]])
>>> array_a * array_b
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
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]])
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
Array([[(Binary(a)*Num(0.2)), (Binary(b)*Num(0.2))],
       [(Binary(c)*Num(0.2)), 0.4]])
```

3.5.3 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)]])
```

3.6 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

3.6.1 UnaryEncInteger

class UnaryEncInteger(*label*, *lower*, *upper*)

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 $2a - b = 1$.

```
>>> from pyqubo import UnaryEncInteger
>>> import dimod
>>> a = UnaryEncInteger("a", 0, 3)
>>> b = UnaryEncInteger("b", 0, 3)
>>> M=2.0
>>> H = (2*a-b-1)**2 + M*(a+b-3)**2
>>> model = H.compile()
>>> q, offset = model.to_qubo()
>>> sampleset = dimod.ExactSolver().sample_qubo(q)
>>> response, broken, e  = model.decode_dimod_response(sampleset, topk=1)[0]
>>> print("a={},b={}".format(sum(response["a"].values()), sum(response["b"].
↪values())))
a=1,b=2
```

3.6.2 LogEncInteger

class LogEncInteger(*label*, *lower*, *upper*)

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()
>>> q, offset = model.to_qubo()
>>> sampleset = dimod.ExactSolver().sample_qubo(q)
>>> response, broken, e  = model.decode_dimod_response(sampleset, topk=1)[0]
>>> sol_a = sum(2**k * v for k, v in response["a"].items())
>>> sol_b = sum(2**k * v for k, v in response["b"].items())
>>> print("a={},b={}".format(sol_a, sol_b))
a=2,b=3
```

3.6.3 OneHotEncInteger

class OneHotEncInteger(*label*, *lower*, *upper*, *strength*)

One-hot encoded integer. The value that takes $[0, n]$ 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()
>>> q, offset = model.to_qubo()
>>> sampleset = dimod.ExactSolver().sample_qubo(q)
>>> solution, broken, e  = model.decode_dimod_response(sampleset, topk=1)[0]
>>> print("a={}".format(1+sum(k*v for k, v in solution["a"].items())))
a=2
```

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*

3.6.4 OrderEncInteger

class OrderEncInteger(*label*, *lower*, *upper*, *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
>>> import dimod
>>> a = OrderEncInteger("a", 0, 3, strength = 5.0)
>>> model = ((a-2)**2).compile()
>>> q, offset = model.to_qubo()
>>> response = dimod.ExactSolver().sample_qubo(q)
>>> solution, broken, e  = model.decode_dimod_response(response, topk=1)[0]
>>> print("a={}".format(sum(solution["a"].values())))
a=2
```

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
>>> import dimod
>>> 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()
>>> q, offset = model.to_qubo()
>>> sampleset = dimod.ExactSolver().sample_qubo(q)
>>> solution, broken, e  = model.decode_dimod_response(sampleset, topk=1)[0]
>>> solution
{'a': {0: 1, 1: 1, 2: 0, 3: 0}, 'b': {0: 1, 1: 1, 2: 0, 3: 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
>>> import dimod
>>> 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()
>>> q, offset = model.to_qubo()
>>> sampleset = dimod.ExactSolver().sample_qubo(q)
>>> solution, broken, e  = model.decode_dimod_response(sampleset, topk=1)[0]
>>> solution
{'a': {0: 1, 1: 1, 2: 0, 3: 0}, 'b': {0: 1, 1: 1, 2: 0, 3: 0}}
```

References

3.7 Logical Constraint

3.7.1 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
```

3.7.2 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
```

3.7.3 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
```

3.7.4 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
```

3.8 Logical Gate

3.8.1 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
```

3.8.2 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
```

3.8.3 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)
>>> 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
```

3.8.4 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
```

3.9 Functions

3.9.1 Sum over indices

class Sum(*start_index*, *end_index*, *func*)
Define sum of the expressions over sequent indices.

Note: Indices run from `start_index` to `end_index-1`.

Parameters

- **start_index** (*int*) – index to start with.
- **end_index** (*int*) – index ends with end_index-1.

- **func** (*function*) – function which takes integer as an argument and returns *Express*.

Example

```
>>> from pyqubo import Sum, Array
>>> x = Array.create('x', 3, 'BINARY')
>>> exp = (Sum(0, 3, lambda i: x[i]) - 1.0)**2
>>> pprint(exp.compile().to_qubo())
({('x[0]', 'x[0]'): -1.0,
  ('x[0]', 'x[1]'): 2.0,
  ('x[0]', 'x[2]'): 2.0,
  ('x[1]', 'x[1]'): -1.0,
  ('x[1]', 'x[2]'): 2.0,
  ('x[2]', 'x[2]'): -1.0},
 1.0)
```

3.10 Utils

3.10.1 Solvers

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

Returns The solution of SA.

Return type dict[label, bit]

```
>>> from pyqubo import Spin, solve_ising
>>> s1, s2, s3 = Spin("s1"), Spin("s2"), Spin("s3")
>>> H = (2*s1 + 4*s2 + 6*s3)**2
>>> model = H.compile()
>>> linear, quad, offset = model.to_ising()
>>> solution = solve_ising(linear, quad)
```

solve_qubo(*qubo*, *num_reads=10*, *sweeps=1000*, *beta_range=(1.0, 50.0)*)
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]

```
>>> from pyqubo import Spin, solve_qubo
>>> s1, s2, s3 = Spin("s1"), Spin("s2"), Spin("s3")
>>> H = (2*s1 + 4*s2 + 6*s3)**2
>>> model = H.compile()
>>> qubo, offset = model.to_qubo()
>>> solution = solve_qubo(qubo)
```

3.10.2 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.

3.11 Internal Class

3.11.1 BinaryProd

class BinaryProd(*keys*)

A product of binary variables. This class is used as a key of dictionary when you represent a polynomial as a dictionary.

For example, a polynomial $2ab + b + 2$ is represented as

```
{BinaryProd({'a', 'b'}): 2.0, BinaryProd({'b'}): 1.0, BinaryProd(set()): 2.0}
```

This class represents product of binary variables. Since $a = a^2 = a^3 = ...$ where a is binary, a product of binary variables can be represented by a set of unique variables. For example, $aabbc$ can be simplified as abc. For this reason, this class contains unique binary variables as a set.

Note: BinaryProd initialized with empty key corresponds to constant.

Parameters **keys** (*set[label]*) – set of variable labels.

calc_product(*dict_values*)

Returns the value of the product of binary variables.

Parameters **dict_values** (*dict[label, float]*) – value of binary variable.

Returns float

is_constant()

Returns whether this is constant or not.

Returns bool

3.11.2 PlaceholderProd

class PlaceholderProd(*keys*)

A product of placeholder variables. This class is used as a key of dictionary when you represent a polynomial as a dictionary.

For example, a polynomial $2a^2b + 2$ is represented as

```
{PlaceholderProd({'a': 2, 'b': 1}): 2.0, PlaceholderProd({}): 2.0}
```

Note: PlaceholderProd initialized with empty key corresponds to constant.

Parameters **keys** (*dict[label, int]*) – dictionary with a key being label, a int value being order of power.

calc_product(*dict_values*)

Returns the value of the product of binary variables.

Parameters **dict_values** (*dict[label, float]*) – value of binary variable.

Returns float

is_constant()

Returns whether this is constant or not.

Returns bool

3.11.3 CompiledQubo

class CompiledQubo(*qubo*, *offset*)

Compiled QUBO.

Parameters

- **qubo** (dict[label, *Coefficient*/float]) – QUBO
- **offset** (*Coefficient*/float) – Offset of QUBO

qubo

QUBO

Type dict[label, *Coefficient*/float]

offset

Offset of QUBO

Type *Coefficient*/float

This contains QUBO and the offset, but the value of the QUBO has not been evaluated yet. To get the final QUBO, you need to evaluate this QUBO by calling `CompiledQubo.eval()`.

evaluate(*feed_dict*)

Returns QUBO where the values are evaluated with feed_dict.

Parameters **feed_dict** (*dict[str, float]*) – The value of *Placeholder*.

Returns `BinaryQuadraticModel`

variables
Unique labels contained in keys of QUBO.

3.11.4 CompiledConstraint

class CompiledConstraint (*polynomial*)
Compiled constraint.

Parameters **polynomial** (dict[*BinaryProd*, float/*Coefficient*]) – Polynomial representation of constraint.

energy (*binary_solution*, *feed_dict*)
Returns the energy of constraint given a solution.

Parameters

- **binary_solution** (*dict[label, bit]*) – Binary solution.
- **feed_dict** (*dict[str, float]*) – The value of placeholders.

Returns Energy of constraint.

Return type float

3.11.5 Coefficient

class Coefficient (*terms*)
The value of QUBO as a function of *Placeholder*.

Energy of QUBO is defined as $E(\mathbf{x}) = \sum_{ij} a_{ij} x_i x_j$.

If the expression contains *Placeholder*, you need to specify the value of placeholders to get the final QUBO. Each coefficient a_{ij} of compiled QUBO is defined as *Coefficient*. If you want to get the final value of a_{ij}, you need to evaluate it with *feed_dict*.

Parameters **terms** (dict[*PlaceholderProd*, float]) – polynomial function of *Placeholder*. The labels in *PlaceholderProd* corresponds to labels of *Placeholder*.

Example

For example, a polynomial $2ab + 2$ is represented as

```
>>> from pyqubo import Coefficient, PlaceholderProd
>>> coeff = Coefficient({PlaceholderProd({'a': 1, 'b': 1}): 2.0, PlaceholderProd(
↪{}): 2.0})
```

If we specify the value of Placeholder as $a = 2, b = 3$, then the evaluated value will be 14.0.

```
>>> coeff.evaluate(feed_dict={'a': 2, 'b': 3})
14.0
```

evaluate (*feed_dict*)
Returns evaluated value with *feed_dict*.

Parameters **feed_dict** (*dict[str, float]*) – The value of placeholder.

Returns float

CHAPTER 4

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

Q

R

S

T

U

V

X