dimod Documentation

Release 0.8.21

D-Wave Systems Inc

Feb 20, 2020

Contents

dimod is a shared API for binary quadratic samplers. It provides a binary quadratic model (BQM) class that contains Ising and quadratic unconstrained binary optimization (QUBO) models used by samplers such as the D-Wave system. It also provides utilities for constructing new samplers and composed samplers and for minor-embedding. Its reference examples include several samplers and composed samplers.

CHAPTER 1

Example Usage

The QUBO form, $\text{E}(a_i, b_{i,j}; q_i) = -q_1 - q_2 + 2q_1q_2$, is related to the Ising form, $\text{E}(h_i, j_{i,j}; s_i) = \frac{1}{2}(s_1s_2 - 1)$, via the simple manipulation $s_i = 2q_i - 1$.

This example constructs a simple QUBO and converts it to Ising format.

```
>>> import dimod
>>> bqm = dimod.BinaryQuadraticModel({0: -1, 1: -1}, {(0, 1): 2}, 0.0, dimod.BINARY)
↪# QUBO
>>> bqm_ising = bqm.change_vartype(dimod.SPIN, inplace=False)  # Ising
```

This example uses one of dimod's test samplers, ExactSolver, a solver that calculates the energies of all possible samples.

```
>>> import dimod
>>> h = {0: 0.0, 1: 0.0}
>>> J = {(0, 1): -1.0}
>>> bqm = dimod.BinaryQuadraticModel.from_ising(h, J)
>>> response = dimod.ExactSolver().sample(bqm)
>>> for sample, energy in response.data(['sample', 'energy']): print(sample, energy)
{0: -1, 1: -1} -1.0
{0: 1, 1: 1} -1.0
{0: 1, 1: -1} 1.0
{0: -1, 1: 1} 1.0
```

CHAPTER 2

Documentation

Note: This documentation is for the latest version of dimod. Documentation for the version currently installed by dwave-ocean-sdk is here: dimod.

2.1 Introduction

dimod provides a binary quadratic model (BQM) class that contains Ising and quadratic unconstrained binary optimization (QUBO) models used, as described in Solving Problems on a D-Wave System, by samplers such as the D-Wave system.

It provides useful functionality for working with these models and samplers; for example *BQM Generators* to build BQMs and *Utilities* for calculating the energy of a sample or serializing dimod objects.

It includes reference *sampler*s and composites for processing binary quadratic programs and refining sampling, and useful for testing your code during development.

It also provides an *API for Samplers and Composites* for constructing new samplers and composed samplers tailored for your problem.

Additionally, it provides some *Higher-Order Composites* and functionality such as reducing higher-order polynomials to BQMs.

2.1.1 Example

Solving problems with large numbers of variables might necessitate the use of decomposition[1] methods such as branch-and-bound to reduce the number of variables. The following illustrative example reduces an Ising model for a small problem (the K4 complete graph), and converts the reduced-variables model to QUBO formulation.

[1] Ocean software's D-Wave Hybrid provides tools for decomposing large problems.

```
>>> import dimod
>>> linear = {1: 1, 2: 2, 3: 3, 4: 4}
>>> quadratic = {(1, 2): 12, (1, 3): 13, (1, 4): 14,
...              (2, 3): 23, (2, 4): 24,
...              (3, 4): 34}
>>> bqm_k4 = dimod.BinaryQuadraticModel(linear, quadratic, 0.5, dimod.SPIN)
>>> bqm_k4.vartype
<Vartype.SPIN: frozenset([1, -1])>
>>> len(bqm_k4.linear)
4
>>> bqm_k4.contract_variables(2, 3)
>>> len(bqm_k4.linear)
3
>>> bqm_no3_qubo = bqm_k4.binary
>>> bqm_no3_qubo.vartype
<Vartype.BINARY: frozenset([0, 1])>
```

2.1.2 Samplers and Composites

*Sampler*s are processes that sample from low-energy states of a problem's objective function. A binary quadratic model (BQM) sampler samples from low-energy states in *model*s such as those defined by an Ising equation or a QUBO problem and returns an iterable of samples, in order of increasing energy. A dimod sampler provides 'sample_qubo' and 'sample_ising' methods as well as the generic BQM sampler method.

*Composed sampler*s apply pre- and/or post-processing to binary quadratic programs without changing the underlying sampler implementation by layering composite patterns on the sampler. For example, a composed sampler might add spin transformations when sampling from the D-Wave system.

*Structured sampler*s are restricted to sampling only binary quadratic models defined on a specific graph.

You can create your own samplers with dimod's `Sampler` abstract base class (ABC) providing complementary methods (e.g., 'sample_qubo' if only 'sample_ising' is implemented), consistent responses, etc.

Examples

This first example uses a composed sampler on the Boolean NOT Gate example detailed in the Getting Started documentation. The `ExactSolver` test sampler calculates the energy of all possible samples; the `FixedVariableComposite` composite sets the value and removes specified variables from the BQM before sending it to the sampler. Fixing variable x, the input to the NOT gate, to 1 results in valid solution $z = 0$ having lower energy (-1) than solution $x = z = 1$, which is an invalid state for a NOT gate.

```
>>> from dimod import FixedVariableComposite, ExactSolver
>>> Q = {('x', 'x'): -1, ('x', 'z'): 2, ('z', 'x'): 0, ('z', 'z'): -1}
>>> composed_sampler = FixedVariableComposite(ExactSolver())
>>> sampleset = composed_sampler.sample_qubo(Q, fixed_variables={'x': 1})
>>> print(sampleset)
   x  z energy num_oc.
0  1  0   -1.0       1
1  1  1    0.0       1
['BINARY', 2 rows, 2 samples, 2 variables]
```

The next example creates a dimod sampler by implementing a single method (in this example the `sample_ising()` method).

```
class LinearIsingSampler(dimod.Sampler):

    def sample_ising(self, h, J):
        sample = linear_ising(h, J)  # Defined elsewhere
        energy = dimod.ising_energy(sample, h, J)
        return dimod.Response.from_samples([sample], {'energy': [energy]})

    @property
    def properties(self):
        return dict()

    @property
    def parameters(self):
        return dict()
```

The `Sampler` ABC provides the other sample methods "for free" as mixins.

2.1.3 Terminology

chain A collection of nodes or variables in the target *graph*/*model* that we want to act as a single node/variable.

chain strength Magnitude of the negative quadratic bias applied between variables to form a *chain*.

composed sampler Samplers that apply pre- and/or post-processing to binary quadratic programs without changing the underlying *sampler* implementation by layering composite patterns on the sampler. For example, a composed sampler might add spin transformations when sampling from the D-Wave system.

graph A collection of nodes and edges. A graph can be derived from a *model*: a node for each variable and an edge for each pair of variables with a non-zero quadratic bias.

model A collection of variables with associated linear and quadratic biases. Sometimes referred to in other tools as a **problem**.

sampler A process that samples from low energy states of a problem's objective function. A binary quadratic model (BQM) sampler samples from low energy states in models such as those defined by an Ising equation or a Quadratic Unconstrained Binary Optimization (QUBO) problem and returns an iterable of samples, in order of increasing energy. A dimod sampler provides 'sample_qubo' and 'sample_ising' methods as well as the generic BQM sampler method.

source In the context of embedding, the model or induced *graph* that we wish to embed. Sometimes referred to in other tools as the **logical** graph/model.

structured sampler Samplers that are restricted to sampling only binary quadratic models defined on a specific *graph*.

target Embedding attempts to create a target *model* from a target *graph*. The process of embedding takes a source model, derives the source graph, maps the source graph to the target graph, then derives the target model. Sometimes referred to in other tools as the **embedded** graph/model.

2.2 Reference Documentation

Release 0.8.21

Date Feb 20, 2020

2.2.1 Binary Quadratic Models

Ising, QUBO, and BQMs

The binary quadratic model (BQM) class contains Ising and quadratic unconstrained binary optimization (QUBO) models used by samplers such as the D-Wave system.

The Ising model is an objective function of N variables $s = [s_1, ..., s_N]$ corresponding to physical Ising spins, where h_i are the biases and $J_{i,j}$ the couplings (interactions) between spins.

$$\text{Ising:} \qquad E(\mathbf{s}|\mathbf{h},\mathbf{J}) = \left\{ \sum_{i=1}^{N} \mathbf{h_i s_i} + \sum_{i<j}^{N} \mathbf{J_{i,j} s_i s_j} \right\} \qquad \mathbf{s_i} \in \{-1,+1\}$$

The QUBO model is an objective function of N binary variables represented as an upper-diagonal matrix Q, where diagonal terms are the linear coefficients and the nonzero off-diagonal terms the quadratic coefficients.

$$\text{QUBO:} \qquad E(\mathbf{x}|\mathbf{Q}) = \sum_{i\leq j}^{N} \mathbf{x_i Q_{i,j} x_j} \qquad \mathbf{x_i} \in \{0,1\}$$

The *BinaryQuadraticModel* class can contain both these models and its methods provide convenient utilities for working with, and interworking between, the two representations of a problem.

These models and their use in solving problems on the D-Wave system is described in the following documentation:

- **Getting Started with the D-Wave System** Introduces key concepts such as objective functions, Ising model, QUBOs, and graphs, explains how these models are used to represent problems, and provides some simple examples.
- **D-Wave Problem-Solving Handbook** Provides a variety of techniques for, and examples of, reformulating problems as BQMs.
- **Solving Problems on a D-Wave System** Describes and demonstrates the use of BQM in the context of Ocean software.

Class

class BinaryQuadraticModel(*linear*, *quadratic*, *offset*, *vartype*, ***kwargs*)

Encodes a binary quadratic model.

Binary quadratic model is the superclass that contains the Ising model and the QUBO.

Parameters

- **linear** (*dict[variable, bias]*) – Linear biases as a dict, where keys are the variables of the binary quadratic model and values the linear biases associated with these variables. A variable can be any python object that is valid as a dictionary key. Biases are generally numbers but this is not explicitly checked.
- **quadratic** (*dict[(variable, variable), bias]*) – Quadratic biases as a dict, where keys are 2-tuples of variables and values the quadratic biases associated with the pair of variables (the interaction). A variable can be any python object that is valid as a dictionary key. Biases are generally numbers but this is not explicitly checked. Interactions that are not unique are added.
- **offset** (*number*) – Constant energy offset associated with the binary quadratic model. Any input type is allowed, but many applications assume that offset is a number. See *BinaryQuadraticModel.energy()*.

- **vartype** (*Vartype*/str/set) – Variable type for the binary quadratic model. Accepted input values:
  - `Vartype.SPIN`, `'SPIN'`, `{-1, 1}`
  - `Vartype.BINARY`, `'BINARY'`, `{0, 1}`
- ****kwargs** – Any additional keyword parameters and their values are stored in *BinaryQuadraticModel.info*.

Notes

The *BinaryQuadraticModel* class does not enforce types on biases and offsets, but most applications that use this class assume that they are numeric.

Examples

This example creates a binary quadratic model with three spin variables.

```
>>> bqm = dimod.BinaryQuadraticModel({0: 1, 1: -1, 2: .5},
...                                  {(0, 1): .5, (1, 2): 1.5},
...                                  1.4,
...                                  dimod.Vartype.SPIN)
```

This example creates a binary quadratic model with non-numeric variables (variables can be any hashable object).

```
>>> bqm = dimod.BQM({'a': 0.0, 'b': -1.0, 'c': 0.5},
...                                  {('a', 'b'): -1.0, ('b', 'c'): 1.5},
...                                  1.4,
...                                  dimod.SPIN)
>>> len(bqm)
3
>>> 'b' in bqm
True
```

linear
Linear biases as a dict, where keys are the variables of the binary quadratic model and values the linear biases associated with these variables.

Type dict[variable, bias]

quadratic
Quadratic biases as a dict, where keys are 2-tuples of variables, which represent an interaction between the two variables, and values are the quadratic biases associated with the interactions.

Type dict[(variable, variable), bias]

offset
The energy offset associated with the model. Same type as given on instantiation.

Type number

vartype
The model's type. One of `Vartype.SPIN` or `Vartype.BINARY`.

Type *Vartype*

variables
The variables in the binary quadratic model as a dictionary keys view object.

Type keysview

adj
The model's interactions as nested dicts. In graphic representation, where variables are nodes and interactions are edges or adjacencies, keys of the outer dict (*adj*) are all the model's nodes (e.g. *v*) and values are the inner dicts. For the inner dict associated with outer-key/node 'v', keys are all the nodes adjacent to *v* (e.g. *u*) and values are quadratic biases associated with the pair of inner and outer keys (*u, v*).

Type dict

info
A place to store miscellaneous data about the binary quadratic model as a whole.

Type dict

SPIN
An alias of `Vartype.SPIN` for easier access.

Type *Vartype*

BINARY
An alias of `Vartype.BINARY` for easier access.

Type *Vartype*

Examples

This example creates an instance of the *BinaryQuadraticModel* class for the K4 complete graph, where the nodes have biases set equal to their sequential labels and interactions are the concatenations of the node pairs (e.g., 23 for u,v = 2,3).

```
>>> import dimod
...
>>> linear = {1: 1, 2: 2, 3: 3, 4: 4}
>>> quadratic = {(1, 2): 12, (1, 3): 13, (1, 4): 14,
...              (2, 3): 23, (2, 4): 24,
...              (3, 4): 34}
>>> offset = 0.0
>>> vartype = dimod.BINARY
>>> bqm_k4 = dimod.BinaryQuadraticModel(linear, quadratic, offset, vartype)
>>> bqm_k4.info = {'Complete K4 binary quadratic model.'}
>>> bqm_k4.info.issubset({'Complete K3 binary quadratic model.',
...                       'Complete K4 binary quadratic model.',
...                       'Complete K5 binary quadratic model.'})
True
>>> bqm_k4.adj.viewitems()   # Show all adjacencies  # doctest: +SKIP
[(1, {2: 12, 3: 13, 4: 14}),
 (2, {1: 12, 3: 23, 4: 24}),
 (3, {1: 13, 2: 23, 4: 34}),
 (4, {1: 14, 2: 24, 3: 34})]
>>> bqm_k4.adj[2]            # Show adjacencies for node 2  # doctest: +SKIP
{1: 12, 3: 23, 4: 24}
>>> bqm_k4.adj[2][3]         # Show the quadratic bias for nodes 2,3 # doctest:␣
↪+SKIP
23
```

Vartype Properties

QUBO (binary-valued variables) and Ising (spin-valued variables) instances of a BQM.

BinaryQuadraticModel.binary	An instance of the QUBO model subclass of the *BinaryQuadraticModel* superclass (a BQM with binary variables).
BinaryQuadraticModel.spin	An instance of the Ising model subclass of the *BinaryQuadraticModel* superclass (a BQM with spin variables).

dimod.BinaryQuadraticModel.binary

BinaryQuadraticModel.**binary**

An instance of the QUBO model subclass of the *BinaryQuadraticModel* superclass (a BQM with binary variables).

Enables access to biases for the binary-valued binary quadratic model regardless of the *vartype* set when the model was created. If the model was created with the *spin* vartype, the QUBO model subclass is instantiated upon the first use of the *binary* property and used in any subsequent reads.

Examples

This example creates an Ising model and uses the *binary* property to instantiate the corresponding QUBO model.

```
>>> import dimod
...
>>> bqm_spin = dimod.BinaryQuadraticModel({0: 0.0, 1: 0.0}, {(0, 1): 0.5}, -0.5,␣
↪dimod.SPIN)
>>> bqm_qubo = bqm_spin.binary
>>> bqm_qubo  # doctest: +SKIP
BinaryQuadraticModel({0: -1.0, 1: -1.0}, {(0, 1): 2.0}, 0.0, Vartype.BINARY)
>>> bqm_qubo.binary is bqm_qubo
True
```

Note: Methods like *add_variable()*, *add_variables_from()*, *add_interaction()*, etc. should only be used on the base model.

Type *BinaryQuadraticModel*

dimod.BinaryQuadraticModel.spin

BinaryQuadraticModel.**spin**

An instance of the Ising model subclass of the *BinaryQuadraticModel* superclass (a BQM with spin variables).

Enables access to biases for the spin-valued binary quadratic model regardless of the *vartype* set when the model was created. If the model was created with the *binary* vartype, the Ising model subclass is instantiated upon the first use of the *spin* property and used in any subsequent reads.

Examples

This example creates a QUBO model and uses the *spin* property to instantiate the corresponding Ising model.

```
>>> import dimod
...
>>> bqm_qubo = dimod.BinaryQuadraticModel({0: -1, 1: -1}, {(0, 1): 2}, 0.0, dimod.
↪BINARY)
>>> bqm_spin = bqm_qubo.spin
>>> bqm_spin   # doctest: +SKIP
BinaryQuadraticModel({0: 0.0, 1: 0.0}, {(0, 1): 0.5}, -0.5, Vartype.SPIN)
>>> bqm_spin.spin is bqm_spin
True
```

Note: Methods like *add_variable()*, *add_variables_from()*, *add_interaction()*, etc. should only be used on the base model.

Type *BinaryQuadraticModel*

Methods

Construction Shortcuts

BinaryQuadraticModel.empty(vartype)	Create an empty binary quadratic model.

dimod.BinaryQuadraticModel.empty

classmethod BinaryQuadraticModel.**empty**(*vartype*)

Create an empty binary quadratic model.

Equivalent to instantiating a *BinaryQuadraticModel* with no bias values and zero offset for the defined *vartype*:

```
BinaryQuadraticModel({}, {}, 0.0, vartype)
```

Parameters **vartype** (*Vartype*/str/set) – Variable type for the binary quadratic model. Accepted input values:

- *Vartype.SPIN*, 'SPIN', {-1, 1}
- *Vartype.BINARY*, 'BINARY', {0, 1}

Examples

```
>>> bqm = dimod.BinaryQuadraticModel.empty(dimod.BINARY)
```

Adding and Removing Variables and Interactions

BinaryQuadraticModel.add_variable(v, bias[, …])	Add variable v and/or its bias to a binary quadratic model.
BinaryQuadraticModel.add_variables_from(linear)	Add variables and/or linear biases to a binary quadratic model.
BinaryQuadraticModel.add_interaction(u, v, bias)	Add an interaction and/or quadratic bias to a binary quadratic model.
BinaryQuadraticModel.add_interactions_from(…)	Add interactions and/or quadratic biases to a binary quadratic model.
BinaryQuadraticModel.add_offset(offset)	Add specified value to the offset of a binary quadratic model.
BinaryQuadraticModel.remove_variable(v)	Remove variable v and all its interactions from a binary quadratic model.
BinaryQuadraticModel.remove_variables_from(…)	Remove specified variables and all of their interactions from a binary quadratic model.
BinaryQuadraticModel.remove_interaction(u, v)	Remove interaction of variables u, v from a binary quadratic model.
BinaryQuadraticModel.remove_interactions_from(…)	Remove all specified interactions from the binary quadratic model.
BinaryQuadraticModel.remove_offset()	Set the binary quadratic model's offset to zero.
BinaryQuadraticModel.update(bqm[, ignore_info])	Update one binary quadratic model from another.

dimod.BinaryQuadraticModel.add_variable

BinaryQuadraticModel.**add_variable**(*v*, *bias*, *vartype=None*)

Add variable v and/or its bias to a binary quadratic model.

Parameters

- **v** (*variable*) – The variable to add to the model. Can be any python object that is a valid dict key.
- **bias** (*bias*) – Linear bias associated with v. If v is already in the model, this value is added to its current linear bias. Many methods and functions expect *bias* to be a number but this is not explicitly checked.
- **vartype** (*Vartype*, optional, default=None) – Vartype of the given bias. If None, the vartype of the binary quadratic model is used. Valid values are `Vartype.SPIN` or `Vartype.BINARY`.

Examples

This example creates an Ising model with two variables, adds a third, and adds to the linear biases of the initial two.

```
>>> import dimod
...
>>> bqm = dimod.BinaryQuadraticModel({0: 0.0, 1: 1.0}, {(0, 1): 0.5}, -0.5, dimod.
↪SPIN)
>>> len(bqm.linear)
2
>>> bqm.add_variable(2, 2.0, vartype=dimod.SPIN)        # Add a new variable
>>> bqm.add_variable(1, 0.33, vartype=dimod.SPIN)
>>> bqm.add_variable(0, 0.33, vartype=dimod.BINARY)     # Binary value is␣
↪converted to spin value
```

(continues on next page)

(continued from previous page)

```
>>> len(bqm.linear)
3
>>> bqm.linear[1]
1.33
```

dimod.BinaryQuadraticModel.add_variables_from

BinaryQuadraticModel.**add_variables_from**(*linear*, *vartype=None*)

Add variables and/or linear biases to a binary quadratic model.

Parameters

- **linear** (*dict[variable, bias]/iterable[(variable, bias)]*) – A collection of variables and their linear biases to add to the model. If a dict, keys are variables in the binary quadratic model and values are biases. Alternatively, an iterable of (variable, bias) pairs. Variables can be any python object that is a valid dict key. Many methods and functions expect the biases to be numbers but this is not explicitly checked. If any variable already exists in the model, its bias is added to the variable's current linear bias.
- **vartype** (*Vartype*, optional, default=None) – Vartype of the given bias. If None, the vartype of the binary quadratic model is used. Valid values are `Vartype.SPIN` or `Vartype.BINARY`.

Examples

This example creates creates an empty Ising model, adds two variables, and subsequently adds to the bias of the one while adding a new, third, variable.

```
>>> import dimod
...
>>> bqm = dimod.BinaryQuadraticModel({}, {}, 0.0, dimod.SPIN)
>>> len(bqm.linear)
0
>>> bqm.add_variables_from({'a': .5, 'b': -1.})
>>> 'b' in bqm
True
>>> bqm.add_variables_from({'b': -1., 'c': 2.0})
>>> bqm.linear['b']
-2.0
```

dimod.BinaryQuadraticModel.add_interaction

BinaryQuadraticModel.**add_interaction**(*u*, *v*, *bias*, *vartype=None*)

Add an interaction and/or quadratic bias to a binary quadratic model.

Parameters

- **v** (*variable*) – One of the pair of variables to add to the model. Can be any python object that is a valid dict key.
- **u** (*variable*) – One of the pair of variables to add to the model. Can be any python object that is a valid dict key.

- **bias** (*bias*) – Quadratic bias associated with u, v. If u, v is already in the model, this value is added to the current quadratic bias. Many methods and functions expect *bias* to be a number but this is not explicitly checked.
- **vartype** (*Vartype*, optional, default=None) – Vartype of the given bias. If None, the vartype of the binary quadratic model is used. Valid values are `Vartype.SPIN` or `Vartype.BINARY`.

Examples

This example creates an Ising model with two variables, adds a third, adds to the bias of the initial interaction, and creates a new interaction.

```
>>> import dimod
...
>>> bqm = dimod.BinaryQuadraticModel({0: 0.0, 1: 1.0}, {(0, 1): 0.5}, -0.5, dimod.
↪SPIN)
>>> len(bqm.quadratic)
1
>>> bqm.add_interaction(0, 2, 2)        # Add new variable 2
>>> bqm.add_interaction(0, 1, .25)
>>> bqm.add_interaction(1, 2, .25, vartype=dimod.BINARY)     # Binary value is␣
↪converted to spin value
>>> len(bqm.quadratic)
3
>>> bqm.quadratic[(0, 1)]
0.75
```

dimod.BinaryQuadraticModel.add_interactions_from

`BinaryQuadraticModel.`**`add_interactions_from`**(*quadratic*, *vartype=None*)

Add interactions and/or quadratic biases to a binary quadratic model.

Parameters

- **quadratic** (*dict[(variable, variable), bias]/ iterable[(variable, variable, bias)]*) – A collection of variables that have an interaction and their quadratic bias to add to the model. If a dict, keys are 2-tuples of variables in the binary quadratic model and values are their corresponding bias. Alternatively, an iterable of 3-tuples. Each interaction in *quadratic* should be unique; that is, if *(u, v)* is a key, *(v, u)* should not be. Variables can be any python object that is a valid dict key. Many methods and functions expect the biases to be numbers but this is not explicitly checked.
- **vartype** (*Vartype*, optional, default=None) – Vartype of the given bias. If None, the vartype of the binary quadratic model is used. Valid values are `Vartype.SPIN` or `Vartype.BINARY`.

Examples

This example creates creates an empty Ising model, adds an interaction for two variables, adds to its bias while adding a new variable, then adds another interaction.

```
>>> import dimod
...
>>> bqm = dimod.BinaryQuadraticModel.empty(dimod.SPIN)
>>> bqm.add_interactions_from({('a', 'b'): -.5})
>>> bqm.quadratic[('a', 'b')]
-0.5
>>> bqm.add_interactions_from({('a', 'b'): -.5, ('a', 'c'): 2})
>>> bqm.add_interactions_from({('b', 'c'): 2}, vartype=dimod.BINARY)   # Binary␣
↪value is converted to spin value
>>> len(bqm.quadratic)
3
>>> bqm.quadratic[('a', 'b')]
-1.0
```

dimod.BinaryQuadraticModel.add_offset

BinaryQuadraticModel.**add_offset**(*offset*)

Add specified value to the offset of a binary quadratic model.

Parameters offset (*number*) – Value to be added to the constant energy offset of the binary quadratic model.

Examples

This example creates an Ising model with an offset of -0.5 and then adds to it.

```
>>> import dimod
...
>>> bqm = dimod.BinaryQuadraticModel({0: 0.0, 1: 0.0}, {(0, 1): 0.5}, -0.5, dimod.
↪SPIN)
>>> bqm.add_offset(1.0)
>>> bqm.offset
0.5
```

dimod.BinaryQuadraticModel.remove_variable

BinaryQuadraticModel.**remove_variable**(*v*)

Remove variable v and all its interactions from a binary quadratic model.

Parameters v (*variable*) – The variable to be removed from the binary quadratic model.

Notes

If the specified variable is not in the binary quadratic model, this function does nothing.

Examples

This example creates an Ising model and then removes one variable.

```
>>> import dimod
...
>>> bqm = dimod.BinaryQuadraticModel({'a': 0.0, 'b': 1.0, 'c': 2.0},
...                                  {('a', 'b'): 0.25, ('a','c'): 0.5, ('b','c'): 0.75}
↪,
...                                  -0.5, dimod.SPIN)
>>> bqm.remove_variable('a')
>>> 'a' in bqm.linear
False
>>> ('b','c') in bqm.quadratic
True
```

dimod.BinaryQuadraticModel.remove_variables_from

BinaryQuadraticModel.**remove_variables_from**(*variables*)

Remove specified variables and all of their interactions from a binary quadratic model.

Parameters **variables** (*iterable*) – A collection of variables to be removed from the binary quadratic model. If any variable is not in the model, it is ignored.

Examples

This example creates an Ising model with three variables and interactions among all of them, and then removes two variables.

```
>>> import dimod
...
>>> bqm = dimod.BinaryQuadraticModel({0: 0.0, 1: 1.0, 2: 2.0},
...                                  {(0, 1): 0.25, (0,2): 0.5, (1,2): 0.75},
...                                  -0.5, dimod.SPIN)
>>> bqm.remove_variables_from([0, 1])
>>> len(bqm.linear)
1
>>> len(bqm.quadratic)
0
```

dimod.BinaryQuadraticModel.remove_interaction

BinaryQuadraticModel.**remove_interaction**(*u*, *v*)

Remove interaction of variables u, v from a binary quadratic model.

Parameters

- **u** (*variable*) – One of the pair of variables in the binary quadratic model that has an interaction.
- **v** (*variable*) – One of the pair of variables in the binary quadratic model that has an interaction.

Notes

Any interaction not in the binary quadratic model is ignored.

Examples

This example creates an Ising model with three variables that has interactions between two, and then removes an interaction.

```
>>> import dimod
...
>>> bqm = dimod.BinaryQuadraticModel({}, {('a', 'b'): -1.0, ('b', 'c'): 1.0}, 0.0,
↪ dimod.SPIN)
>>> bqm.remove_interaction('b', 'c')
>>> ('b', 'c') in bqm.quadratic
False
>>> bqm.remove_interaction('a', 'c')  # not an interaction, so ignored
>>> len(bqm.quadratic)
1
```

dimod.BinaryQuadraticModel.remove_interactions_from

BinaryQuadraticModel.**remove_interactions_from**(*interactions*)

Remove all specified interactions from the binary quadratic model.

Parameters interactions (*iterable[[variable, variable]]*) – A collection of interactions. Each interaction should be a 2-tuple of variables in the binary quadratic model.

Notes

Any interaction not in the binary quadratic model is ignored.

Examples

This example creates an Ising model with three variables that has interactions between two, and then removes an interaction.

```
>>> import dimod
...
>>> bqm = dimod.BinaryQuadraticModel({}, {('a', 'b'): -1.0, ('b', 'c'): 1.0}, 0.0,
↪ dimod.SPIN)
>>> bqm.remove_interactions_from([('b', 'c'), ('a', 'c')])  # ('a', 'c') is not␣
↪an interaction, so ignored
>>> len(bqm.quadratic)
1
```

dimod.BinaryQuadraticModel.remove_offset

BinaryQuadraticModel.**remove_offset**()

Set the binary quadratic model's offset to zero.

Examples

This example creates an Ising model with a positive energy offset, and then removes it.

```
>>> import dimod
...
>>> bqm = dimod.BinaryQuadraticModel({}, {}, 1.3, dimod.SPIN)
>>> bqm.remove_offset()
>>> bqm.offset
0.0
```

dimod.BinaryQuadraticModel.update

BinaryQuadraticModel.**update**(*bqm*, *ignore_info=True*)

Update one binary quadratic model from another.

Parameters

- **bqm** (*BinaryQuadraticModel*) – The updating binary quadratic model. Any variables in the updating model are added to the updated model. Values of biases and the offset in the updating model are added to the corresponding values in the updated model.
- **ignore_info** (*bool, optional, default=True*) – If True, info in the given binary quadratic model is ignored, otherwise *BinaryQuadraticModel.info* is updated with the given binary quadratic model's info, potentially overwriting values.

Examples

This example creates two binary quadratic models and updates the first from the second.

```
>>> import dimod
...
>>> linear1 = {1: 1, 2: 2}
>>> quadratic1 = {(1, 2): 12}
>>> bqm1 = dimod.BinaryQuadraticModel(linear1, quadratic1, 0.5, dimod.SPIN)
>>> bqm1.info = {'BQM number 1'}
>>> linear2 = {2: 0.25, 3: 0.35}
>>> quadratic2 = {(2, 3): 23}
>>> bqm2 = dimod.BinaryQuadraticModel(linear2, quadratic2, 0.75, dimod.SPIN)
>>> bqm2.info = {'BQM number 2'}
>>> bqm1.update(bqm2)
>>> bqm1.offset
1.25
>>> 'BQM number 2' in bqm1.info
False
>>> bqm1.update(bqm2, ignore_info=False)
>>> 'BQM number 2' in bqm1.info
True
>>> bqm1.offset
2.0
```

Transformations

BinaryQuadraticModel.contract_variables(u, v)	Enforce u, v being the same variable in a binary quadratic model.

Continued on next page

Table 4 – continued from previous page

BinaryQuadraticModel.fix_variable(v, value)	Fix the value of a variable and remove it from a binary quadratic model.
BinaryQuadraticModel.fix_variables(fixed)	Fix the value of the variables and remove it from a binary quadratic model.
BinaryQuadraticModel.flip_variable(v)	Flip variable v in a binary quadratic model.
BinaryQuadraticModel.normalize([bias_range, ...])	Normalizes the biases of the binary quadratic model such that they fall in the provided range(s), and adjusts the offset appropriately.
BinaryQuadraticModel.relabel_variables(mapping)	Relabel variables of a binary quadratic model as specified by mapping.
BinaryQuadraticModel.scale(scalar[, ...])	Multiply by the specified scalar all the biases and offset of a binary quadratic model.

dimod.BinaryQuadraticModel.contract_variables

BinaryQuadraticModel.**contract_variables**(*u*, *v*)

Enforce u, v being the same variable in a binary quadratic model.

The resulting variable is labeled 'u'. Values of interactions between *v* and variables that *u* interacts with are added to the corresponding interactions of *u*.

Parameters

- **u** (*variable*) – Variable in the binary quadratic model.
- **v** (*variable*) – Variable in the binary quadratic model.

Examples

This example creates a binary quadratic model representing the K4 complete graph and contracts node (variable) 3 into node 2. The interactions between 3 and its neighbors 1 and 4 are added to the corresponding interactions between 2 and those same neighbors.

```
>>> import dimod
...
>>> linear = {1: 1, 2: 2, 3: 3, 4: 4}
>>> quadratic = {(1, 2): 12, (1, 3): 13, (1, 4): 14,
...              (2, 3): 23, (2, 4): 24,
...              (3, 4): 34}
>>> bqm = dimod.BinaryQuadraticModel(linear, quadratic, 0.5, dimod.SPIN)
>>> bqm.contract_variables(2, 3)
>>> 3 in bqm.linear
False
>>> bqm.quadratic[(1, 2)]
25
```

dimod.BinaryQuadraticModel.fix_variable

BinaryQuadraticModel.**fix_variable**(*v*, *value*)

Fix the value of a variable and remove it from a binary quadratic model.

Parameters

- **v** (*variable*) – Variable in the binary quadratic model to be fixed.

- **value** (*int*) – Value assigned to the variable. Values must match the *Vartype* of the binary quadratic model.

Examples

This example creates a binary quadratic model with one variable and fixes its value.

```
>>> import dimod
...
>>> bqm = dimod.BinaryQuadraticModel({'a': -.5, 'b': 0.}, {('a', 'b'): -1}, 0.0,␣
↪dimod.SPIN)
>>> bqm.fix_variable('a', -1)
>>> bqm.offset
0.5
>>> bqm.linear['b']
1.0
>>> 'a' in bqm
False
```

dimod.BinaryQuadraticModel.fix_variables

BinaryQuadraticModel.**fix_variables**(*fixed*)
Fix the value of the variables and remove it from a binary quadratic model.

Parameters **fixed** (*dict*) – A dictionary of variable assignments.

Examples

```
>>> bqm = dimod.BinaryQuadraticModel({'a': -.5, 'b': 0., 'c': 5}, {('a', 'b'): -1}
↪, 0.0, dimod.SPIN)
>>> bqm.fix_variables({'a': -1, 'b': +1})
```

dimod.BinaryQuadraticModel.flip_variable

BinaryQuadraticModel.**flip_variable**(*v*)
Flip variable v in a binary quadratic model.

Parameters **v** (*variable*) – Variable in the binary quadratic model. If v is not in the binary quadratic model, it is ignored.

Examples

This example creates a binary quadratic model with two variables and inverts the value of one.

```
>>> import dimod
...
>>> bqm = dimod.BinaryQuadraticModel({1: 1, 2: 2}, {(1, 2): 0.5}, 0.5, dimod.SPIN)
>>> bqm.flip_variable(1)
>>> bqm.linear[1], bqm.linear[2], bqm.quadratic[(1, 2)]
(-1.0, 2, -0.5)
```

dimod.BinaryQuadraticModel.normalize

BinaryQuadraticModel.**normalize**(*bias_range=1*, *quadratic_range=None*, *ignored_variables=None*, *ignored_interactions=None*, *ignore_offset=False*)

Normalizes the biases of the binary quadratic model such that they fall in the provided range(s), and adjusts the offset appropriately.

If *quadratic_range* is provided, then *bias_range* will be treated as the range for the linear biases and *quadratic_range* will be used for the range of the quadratic biases.

Parameters

- **bias_range** (*number/pair*) – Value/range by which to normalize the all the biases, or if *quadratic_range* is provided, just the linear biases.
- **quadratic_range** (*number/pair*) – Value/range by which to normalize the quadratic biases.
- **ignored_variables** (*iterable, optional*) – Biases associated with these variables are not scaled.
- **ignored_interactions** (*iterable[tuple], optional*) – As an iterable of 2-tuples. Biases associated with these interactions are not scaled.
- **ignore_offset** (*bool, default=False*) – If True, the offset is not scaled.

Examples

```
>>> bqm = dimod.BinaryQuadraticModel({'a': -2.0, 'b': 1.5},
...                                  {('a', 'b'): -1.0},
...                                  1.0, dimod.SPIN)
>>> max(abs(bias) for bias in bqm.linear.values())
2.0
>>> max(abs(bias) for bias in bqm.quadratic.values())
1.0
>>> bqm.normalize([-1.0, 1.0])
>>> max(abs(bias) for bias in bqm.linear.values())
1.0
>>> max(abs(bias) for bias in bqm.quadratic.values())
0.5
```

dimod.BinaryQuadraticModel.relabel_variables

BinaryQuadraticModel.**relabel_variables**(*mapping*, *inplace=True*)

Relabel variables of a binary quadratic model as specified by mapping.

Parameters

- **mapping** (*dict*) – Dict mapping current variable labels to new ones. If an incomplete mapping is provided, unmapped variables retain their current labels.
- **inplace** (*bool, optional, default=True*) – If True, the binary quadratic model is updated in-place; otherwise, a new binary quadratic model is returned.

Returns A binary quadratic model with the variables relabeled. If *inplace* is set to True, returns itself.

Return type *BinaryQuadraticModel*

Examples

This example creates a binary quadratic model with two variables and relables one.

```
>>> import dimod
...
>>> model = dimod.BinaryQuadraticModel({0: 0., 1: 1.}, {(0, 1): -1}, 0.0,␣
↪vartype=dimod.SPIN)
>>> model.relabel_variables({0: 'a'})    # doctest: +SKIP
BinaryQuadraticModel({1: 1.0, 'a': 0.0}, {('a', 1): -1}, 0.0, Vartype.SPIN)
```

This example creates a binary quadratic model with two variables and returns a new model with relabled variables.

```
>>> import dimod
...
>>> model = dimod.BinaryQuadraticModel({0: 0., 1: 1.}, {(0, 1): -1}, 0.0,␣
↪vartype=dimod.SPIN)
>>> new_model = model.relabel_variables({0: 'a', 1: 'b'}, inplace=False)   #␣
↪doctest: +SKIP
>>> new_model.quadratic       # doctest: +SKIP
{('a', 'b'): -1}
```

dimod.BinaryQuadraticModel.scale

`BinaryQuadraticModel.`**`scale`**(*scalar*, *ignored_variables=None*, *ignored_interactions=None*, *ignore_offset=False*)

Multiply by the specified scalar all the biases and offset of a binary quadratic model.

Parameters

- **`scalar`** (*`number`*) – Value by which to scale the energy range of the binary quadratic model.
- **`ignored_variables`** (*`iterable, optional`*) – Biases associated with these variables are not scaled.
- **`ignored_interactions`** (*`iterable[tuple], optional`*) – As an iterable of 2-tuples. Biases associated with these interactions are not scaled.
- **`ignore_offset`** (*`bool, default=False`*) – If True, the offset is not scaled.

Examples

This example creates a binary quadratic model and then scales it to half the original energy range.

```
>>> import dimod
...
>>> bqm = dimod.BinaryQuadraticModel({'a': -2.0, 'b': 2.0}, {('a', 'b'): -1.0}, 1.
↪0, dimod.SPIN)
>>> bqm.scale(0.5)
>>> bqm.linear['a']
-1.0
```

(continues on next page)

(continued from previous page)

```
>>> bqm.quadratic[('a', 'b')]
-0.5
>>> bqm.offset
0.5
```

Change Vartype

BinaryQuadraticModel.change_vartype(vartype)	Create a binary quadratic model with the specified vartype.

dimod.BinaryQuadraticModel.change_vartype

BinaryQuadraticModel.**change_vartype**(*vartype*, *inplace=True*)

Create a binary quadratic model with the specified vartype.

Parameters

- **vartype** (*Vartype*/str/set, optional) – Variable type for the changed model. Accepted input values:
  - Vartype.SPIN, 'SPIN', {-1, 1}
  - Vartype.BINARY, 'BINARY', {0, 1}
- **inplace** (*bool, optional, default=True*) – If True, the binary quadratic model is updated in-place; otherwise, a new binary quadratic model is returned.

Returns *BinaryQuadraticModel*. A new binary quadratic model with vartype matching input 'vartype'.

Examples

This example creates an Ising model and then creates a QUBO from it.

```
>>> import dimod
...
>>> bqm_spin = dimod.BinaryQuadraticModel({1: 1, 2: 2}, {(1, 2): 0.5}, 0.5, dimod.
↪SPIN)
>>> bqm_qubo = bqm_spin.change_vartype('BINARY', inplace=False)
>>> bqm_spin.offset, bqm_spin.vartype
(0.5, <Vartype.SPIN: frozenset({1, -1})>)
>>> bqm_qubo.offset, bqm_qubo.vartype
(-2.0, <Vartype.BINARY: frozenset({0, 1})>)
```

Copy

BinaryQuadraticModel.copy()	Create a copy of a binary quadratic model.

dimod.BinaryQuadraticModel.copy

BinaryQuadraticModel.**copy**()

Create a copy of a binary quadratic model.

Returns *BinaryQuadraticModel*

Examples

```
>>> bqm = dimod.BinaryQuadraticModel({1: 1, 2: 2}, {(1, 2): 0.5}, 0.5, dimod.SPIN)
>>> bqm2 = bqm.copy()
```

Energy

BinaryQuadraticModel.energy(sample)	Determine the energy of the specified sample of a binary quadratic model.
BinaryQuadraticModel.energies(samples_like)	Determine the energies of the given samples.

dimod.BinaryQuadraticModel.energy

BinaryQuadraticModel.**energy**(*sample*)

Determine the energy of the specified sample of a binary quadratic model.

Energy of a sample for a binary quadratic model is defined as a sum, offset by the constant energy offset associated with the binary quadratic model, of the sample multipled by the linear bias of the variable and all its interactions; that is,

$$E(\mathbf{s}) = \sum_v h_v s_v + \sum_{u,v} J_{u,v} s_u s_v + c$$

where s_v is the sample, h_v is the linear bias, $J_{u,v}$ the quadratic bias (interactions), and c the energy offset.

Code for the energy calculation might look like the following:

```
energy = model.offset  # doctest: +SKIP
for v in model:  # doctest: +SKIP
    energy += model.linear[v] * sample[v]
for u, v in model.quadratic:  # doctest: +SKIP
    energy += model.quadratic[(u, v)] * sample[u] * sample[v]
```

Parameters **sample** (*dict*) – Sample for which to calculate the energy, formatted as a dict where keys are variables and values are the value associated with each variable.

Returns Energy for the sample.

Return type float

Examples

This example creates a binary quadratic model and returns the energies for a couple of samples.

```
>>> import dimod
>>> bqm = dimod.BinaryQuadraticModel({1: 1, 2: 1}, {(1, 2): 1}, 0.5, dimod.SPIN)
>>> bqm.energy({1: -1, 2: -1})
-0.5
>>> bqm.energy({1: 1, 2: 1})
3.5
```

dimod.BinaryQuadraticModel.energies

BinaryQuadraticModel.**energies**(*samples_like*, *dtype=<class 'float'>*)

Determine the energies of the given samples.

Parameters

- **samples_like** (*samples_like*) – A collection of raw samples. *samples_like* is an extension of NumPy's array_like structure. See *as_samples()*.
- **dtype** (numpy.dtype) – The data type of the returned energies.

Returns The energies.

Return type numpy.ndarray

Converting To and From Other Formats

BinaryQuadraticModel.from_coo(obj[, vartype])	Deserialize a binary quadratic model from a COOrdinate format encoding.
BinaryQuadraticModel.from_ising(h, J[, offset])	Create a binary quadratic model from an Ising problem.
BinaryQuadraticModel.from_networkx_graph(G)	Create a binary quadratic model from a NetworkX graph.
BinaryQuadraticModel.from_numpy_matrix(mat)	Create a binary quadratic model from a NumPy array.
BinaryQuadraticModel.from_numpy_vectors(…)	Create a binary quadratic model from vectors.
BinaryQuadraticModel.from_qubo(Q[, offset])	Create a binary quadratic model from a QUBO model.
BinaryQuadraticModel.from_pandas_dataframe(bqm_df)	Create a binary quadratic model from a QUBO model formatted as a pandas DataFrame.
BinaryQuadraticModel.from_serializable(obj)	Deserialize a binary quadratic model.
BinaryQuadraticModel.to_coo([fp, vartype_header])	Serialize the binary quadratic model to a COOrdinate format encoding.
BinaryQuadraticModel.to_ising()	Converts a binary quadratic model to Ising format.
BinaryQuadraticModel.to_networkx_graph([…])	Convert a binary quadratic model to NetworkX graph format.
BinaryQuadraticModel.to_numpy_matrix([…])	Convert a binary quadratic model to NumPy 2D array.
BinaryQuadraticModel.to_numpy_vectors([…])	Convert a binary quadratic model to numpy arrays.
BinaryQuadraticModel.to_qubo()	Convert a binary quadratic model to QUBO format.

Continued on next page

Table 8 – continued from previous page

BinaryQuadraticModel.to_pandas_dataframe()	Convert a binary quadratic model to pandas DataFrame format.
BinaryQuadraticModel.to_serializable([...])	Convert the binary quadratic model to a serializable object.

dimod.BinaryQuadraticModel.from_coo

classmethod BinaryQuadraticModel.**from_coo**(*obj*, *vartype=None*)

Deserialize a binary quadratic model from a COOrdinate format encoding.

Parameters

- **obj** – (str/file): Either a string or a *.read()*-supporting file object that represents linear and quadratic biases for a binary quadratic model. This data is stored as a list of 3-tuples, (i, j, bias), where $i = j$ for linear biases.
- **vartype** (*Vartype*/str/set, optional) – Variable type for the binary quadratic model. Accepted input values:
  - `Vartype.SPIN`, `'SPIN'`, `{-1, 1}`
  - `Vartype.BINARY`, `'BINARY'`, `{0, 1}`

  If not provided, the vartype must be specified with a header in the file.

Note: Variables must use index lables (numeric lables). Binary quadratic models created from COOrdinate format encoding have offsets set to zero.

Examples

An example of a binary quadratic model encoded in COOrdinate format.

```
0 0 0.50000
0 1 0.50000
1 1 -1.50000
```

The Coordinate format with a header

```
# vartype=SPIN
0 0 0.50000
0 1 0.50000
1 1 -1.50000
```

This example saves a binary quadratic model to a COOrdinate-format file and creates a new model by reading the saved file.

```
>>> import dimod
>>> bqm = dimod.BinaryQuadraticModel({0: -1.0, 1: 1.0}, {(0, 1): -1.0}, 0.0, dimod.BINARY)
>>> with open('tmp.qubo', 'w') as file:      # doctest: +SKIP
...     bqm.to_coo(file)
>>> with open('tmp.qubo', 'r') as file:      # doctest: +SKIP
...     new_bqm = dimod.BinaryQuadraticModel.from_coo(file, dimod.BINARY)
>>> any(new_bqm)        # doctest: +SKIP
True
```

dimod.BinaryQuadraticModel.from_ising

classmethod BinaryQuadraticModel.**from_ising**(*h*, *J*, *offset=0.0*)

Create a binary quadratic model from an Ising problem.

Parameters

- **h** (*dict/list*) – Linear biases of the Ising problem. If a dict, should be of the form *{v: bias, …}* where v is a spin-valued variable and *bias* is its associated bias. If a list, it is treated as a list of biases where the indices are the variable labels.
- **J** (*dict[(variable, variable), bias]*) – Quadratic biases of the Ising problem.
- **offset** (*optional, default=0.0*) – Constant offset applied to the model.

Returns Binary quadratic model with vartype set to `Vartype.SPIN`.

Return type *BinaryQuadraticModel*

Examples

This example creates a binary quadratic model from an Ising problem.

```
>>> import dimod
>>> h = {1: 1, 2: 2, 3: 3, 4: 4}
>>> J = {(1, 2): 12, (1, 3): 13, (1, 4): 14,
...      (2, 3): 23, (2, 4): 24,
...      (3, 4): 34}
>>> model = dimod.BinaryQuadraticModel.from_ising(h, J, offset = 0.0)
>>> model      # doctest: +SKIP
BinaryQuadraticModel({1: 1, 2: 2, 3: 3, 4: 4}, {(1, 2): 12, (1, 3): 13, (1, 4):
→14, (2, 3): 23, (3, 4): 34, (2, 4): 24}, 0.0, Vartype.SPIN)
```

dimod.BinaryQuadraticModel.from_networkx_graph

classmethod BinaryQuadraticModel.**from_networkx_graph**(*G*, *vartype=None*, *node_attribute_name='bias'*, *edge_attribute_name='bias'*)

Create a binary quadratic model from a NetworkX graph.

Parameters

- **G** (`networkx.Graph`) – A NetworkX graph with biases stored as node/edge attributes.
- **vartype** (*Vartype*/str/set, optional) – Variable type for the binary quadratic model. Accepted input values:
  - `Vartype.SPIN`, `'SPIN'`, `{-1, 1}`
  - `Vartype.BINARY`, `'BINARY'`, `{0, 1}`

  If not provided, the *G* should have a vartype attribute. If *vartype* is provided and *G.vartype* exists then the argument overrides the property.
- **node_attribute_name** (*hashable, optional, default='bias'*) – Attribute name for linear biases. If the node does not have a matching attribute then the bias defaults to 0.

- **edge_attribute_name** (*hashable, optional, default='bias'*) – Attribute name for quadratic biases. If the edge does not have a matching attribute then the bias defaults to 0.

Returns *BinaryQuadraticModel*

Examples

```
>>> import networkx as nx
...
>>> G = nx.Graph()
>>> G.add_node('a', bias=.5)
>>> G.add_edge('a', 'b', bias=-1)
>>> bqm = dimod.BinaryQuadraticModel.from_networkx_graph(G, 'SPIN')
>>> bqm.adj['a']['b']
-1
```

dimod.BinaryQuadraticModel.from_numpy_matrix

classmethod BinaryQuadraticModel.**from_numpy_matrix**(*mat*, *variable_order=None*, *offset=0.0*, *interactions=None*)

Create a binary quadratic model from a NumPy array.

Parameters

- **mat** (numpy.ndarray) – Coefficients of a quadratic unconstrained binary optimization (QUBO) model formatted as a square NumPy 2D array.
- **variable_order** (*list, optional*) – If provided, labels the QUBO variables; otherwise, row/column indices are used. If *variable_order* is longer than the array, extra values are ignored.
- **offset** (*optional, default=0.0*) – Constant offset for the binary quadratic model.
- **interactions** (*iterable, optional, default=[]*) – Any additional 0.0-bias interactions to be added to the binary quadratic model.

Returns Binary quadratic model with vartype set to `Vartype.BINARY`.

Return type *BinaryQuadraticModel*

Examples

This example creates a binary quadratic model from a QUBO in NumPy format while adding an interaction with a new variable ('f'), ignoring an extra variable ('g'), and setting an offset.

```
>>> import dimod
>>> import numpy as np
>>> Q = np.array([[1, 0, 0, 10, 11],
...               [0, 2, 0, 12, 13],
...               [0, 0, 3, 14, 15],
...               [0, 0, 0, 4, 0],
...               [0, 0, 0, 0, 5]]).astype(np.float32)
>>> model = dimod.BinaryQuadraticModel.from_numpy_matrix(Q,
...         variable_order = ['a', 'b', 'c', 'd', 'e', 'f', 'g'],
...         offset = 2.5,
```

(continues on next page)

(continued from previous page)

```
...         interactions = {('a', 'f')})
>>> model.linear   # doctest: +SKIP
{'a': 1.0, 'b': 2.0, 'c': 3.0, 'd': 4.0, 'e': 5.0, 'f': 0.0}
>>> model.quadratic[('a', 'd')]
10.0
>>> model.quadratic[('a', 'f')]
0.0
>>> model.offset
2.5
```

dimod.BinaryQuadraticModel.from_numpy_vectors

classmethod BinaryQuadraticModel.**from_numpy_vectors**(*linear*, *quadratic*, *offset*, *vartype*, *variable_order=None*)

Create a binary quadratic model from vectors.

Parameters

- **linear** (*array_like*) – A 1D array-like iterable of linear biases.
- **quadratic** (*tuple[array_like, array_like, array_like]*) – A 3-tuple of 1D array_like vectors of the form (row, col, bias).
- **offset** (*numeric, optional*) – Constant offset for the binary quadratic model.
- **vartype** (*Vartype*/str/set) – Variable type for the binary quadratic model. Accepted input values:
  - `Vartype.SPIN`, `'SPIN'`, `{-1, 1}`
  - `Vartype.BINARY`, `'BINARY'`, `{0, 1}`
- **variable_order** (*iterable, optional*) – If provided, labels the variables; otherwise, indices are used.

Returns *BinaryQuadraticModel*

Examples

```
>>> import dimod
>>> import numpy as np
...
>>> linear_vector = np.asarray([-1, 1])
>>> quadratic_vectors = (np.asarray([0]), np.asarray([1]), np.asarray([-1.0]))
>>> bqm = dimod.BinaryQuadraticModel.from_numpy_vectors(linear_vector, quadratic_
↪vectors, 0.0, dimod.SPIN)
>>> print(bqm.quadratic)
{(0, 1): -1.0}
```

dimod.BinaryQuadraticModel.from_qubo

classmethod BinaryQuadraticModel.**from_qubo**(*Q*, *offset=0.0*)

Create a binary quadratic model from a QUBO model.

Parameters

- **Q** (*dict*) – Coefficients of a quadratic unconstrained binary optimization (QUBO) problem. Should be a dict of the form *{(u, v): bias, . . . }* where *u*, *v*, are binary-valued variables and *bias* is their associated coefficient.
- **offset** (*optional, default=0.0*) – Constant offset applied to the model.

Returns Binary quadratic model with vartype set to `Vartype.BINARY`.

Return type *BinaryQuadraticModel*

Examples

This example creates a binary quadratic model from a QUBO model.

```
>>> import dimod
>>> Q = {(0, 0): -1, (1, 1): -1, (0, 1): 2}
>>> model = dimod.BinaryQuadraticModel.from_qubo(Q, offset = 0.0)
>>> model.linear    # doctest: +SKIP
{0: -1, 1: -1}
>>> model.vartype
<Vartype.BINARY: frozenset({0, 1})>
```

dimod.BinaryQuadraticModel.from_pandas_dataframe

classmethod `BinaryQuadraticModel.`**from_pandas_dataframe**(*bqm_df, offset=0.0, interactions=None*)

Create a binary quadratic model from a QUBO model formatted as a pandas DataFrame.

Parameters

- **bqm_df** (`pandas.DataFrame`) – Quadratic unconstrained binary optimization (QUBO) model formatted as a pandas DataFrame. Row and column indices label the QUBO variables; values are QUBO coefficients.
- **offset** (*optional, default=0.0*) – Constant offset for the binary quadratic model.
- **interactions** (*iterable, optional, default=[]*) – Any additional 0.0-bias interactions to be added to the binary quadratic model.

Returns Binary quadratic model with vartype set to `vartype.BINARY`.

Return type *BinaryQuadraticModel*

Examples

This example creates a binary quadratic model from a QUBO in pandas DataFrame format while adding an interaction and setting a constant offset.

```
>>> import dimod
>>> import pandas as pd
>>> pd_qubo = pd.DataFrame(data={0: [-1, 0], 1: [2, -1]})
>>> pd_qubo
   0  1
0 -1  2
1  0 -1
>>> model = dimod.BinaryQuadraticModel.from_pandas_dataframe(pd_qubo,
...         offset = 2.5,
```

(continues on next page)

(continued from previous page)

```
...         interactions = {(0,2), (1,2)})
>>> model.linear        # doctest: +SKIP
{0: -1, 1: -1.0, 2: 0.0}
>>> model.quadratic     # doctest: +SKIP
{(0, 1): 2, (0, 2): 0.0, (1, 2): 0.0}
>>> model.offset
2.5
>>> model.vartype
<Vartype.BINARY: frozenset({0, 1})>
```

dimod.BinaryQuadraticModel.from_serializable

classmethod BinaryQuadraticModel.**from_serializable**(*obj*)

Deserialize a binary quadratic model.

Parameters **obj** (*dict*) – A binary quadratic model serialized by *to_serializable()*.

Returns *BinaryQuadraticModel*

Examples

Encode and decode using JSON

```
>>> import dimod
>>> import json
...
>>> bqm = dimod.BinaryQuadraticModel({'a': -1.0, 'b': 1.0}, {('a', 'b'): -1.0}, 0.
↪0, dimod.SPIN)
>>> s = json.dumps(bqm.to_serializable())
>>> new_bqm = dimod.BinaryQuadraticModel.from_serializable(json.loads(s))
```

See also:

to_serializable()

json.loads(), json.load() JSON deserialization functions

dimod.BinaryQuadraticModel.to_coo

BinaryQuadraticModel.**to_coo**(*fp=None*, *vartype_header=False*)

Serialize the binary quadratic model to a COOrdinate format encoding.

Parameters

- **fp** (*file, optional*) – *.write()*-supporting file object to save the linear and quadratic biases of a binary quadratic model to. The model is stored as a list of 3-tuples, (i, j, bias), where $i = j$ for linear biases. If not provided, returns a string.
- **vartype_header** (*bool, optional, default=False*) – If true, the binary quadratic model's variable type as prepended to the string or file as a header.

Note: Variables must use index lables (numeric lables). Binary quadratic models saved to COOrdinate format encoding do not preserve offsets.

Examples

This is an example of a binary quadratic model encoded in COOrdinate format.

```
0 0 0.50000
0 1 0.50000
1 1 -1.50000
```

The Coordinate format with a header

```
# vartype=SPIN
0 0 0.50000
0 1 0.50000
1 1 -1.50000
```

This is an example of writing a binary quadratic model to a COOrdinate-format file.

```
>>> bqm = dimod.BinaryQuadraticModel({0: -1.0, 1: 1.0}, {(0, 1): -1.0}, 0.0,␣
↪dimod.SPIN)
>>> with open('tmp.ising', 'w') as file:  # doctest: +SKIP
...     bqm.to_coo(file)
```

This is an example of writing a binary quadratic model to a COOrdinate-format string.

```
>>> bqm = dimod.BinaryQuadraticModel({0: -1.0, 1: 1.0}, {(0, 1): -1.0}, 0.0,␣
↪dimod.SPIN)
>>> bqm.to_coo()  # doctest: +SKIP
0 0 -1.000000
0 1 -1.000000
1 1 1.000000
```

dimod.BinaryQuadraticModel.to_ising

`BinaryQuadraticModel.`**`to_ising`**`()`

Converts a binary quadratic model to Ising format.

If the binary quadratic model's vartype is not `Vartype.SPIN`, values are converted.

Returns 3-tuple of form (*linear*, *quadratic*, *offset*), where *linear* is a dict of linear biases, *quadratic* is a dict of quadratic biases, and *offset* is a number that represents the constant offset of the binary quadratic model.

Return type tuple

Examples

This example converts a binary quadratic model to an Ising problem.

```
>>> import dimod
>>> model = dimod.BinaryQuadraticModel({0: 1, 1: -1, 2: .5},
...                                    {(0, 1): .5, (1, 2): 1.5},
...                                    1.4,
...                                    dimod.SPIN)
>>> model.to_ising()    # doctest: +SKIP
({0: 1, 1: -1, 2: 0.5}, {(0, 1): 0.5, (1, 2): 1.5}, 1.4)
```

dimod.BinaryQuadraticModel.to_networkx_graph

BinaryQuadraticModel.**to_networkx_graph**(*node_attribute_name='bias'*, *edge_attribute_name='bias'*)

Convert a binary quadratic model to NetworkX graph format.

Parameters

- **node_attribute_name** (*hashable, optional, default='bias'*) – Attribute name for linear biases.
- **edge_attribute_name** (*hashable, optional, default='bias'*) – Attribute name for quadratic biases.

Returns A NetworkX graph with biases stored as node/edge attributes.

Return type networkx.Graph

Examples

This example converts a binary quadratic model to a NetworkX graph, using first the default attribute name for quadratic biases then "weight".

```
>>> import networkx as nx
>>> bqm = dimod.BinaryQuadraticModel({0: 1, 1: -1, 2: .5},
...                                  {(0, 1): .5, (1, 2): 1.5},
...                                  1.4,
...                                  dimod.SPIN)
>>> BQM = bqm.to_networkx_graph()
>>> BQM[0][1]['bias']
0.5
>>> BQM.node[0]['bias']
1
>>> BQM_w = bqm.to_networkx_graph(edge_attribute_name='weight')
>>> BQM_w[0][1]['weight']
0.5
```

dimod.BinaryQuadraticModel.to_numpy_matrix

BinaryQuadraticModel.**to_numpy_matrix**(*variable_order=None*)

Convert a binary quadratic model to NumPy 2D array.

Parameters **variable_order** (*list, optional*) – If provided, indexes the rows/columns of the NumPy array. If *variable_order* includes any variables not in the binary quadratic model, these are added to the NumPy array.

Returns The binary quadratic model as a NumPy 2D array. Note that the binary quadratic model is converted to BINARY vartype.

Return type numpy.ndarray

Notes

The matrix representation of a binary quadratic model only makes sense for binary models. For a binary sample x, the energy of the model is given by:

$$E(x) = x^T Q x$$

The offset is dropped when converting to a NumPy array.

Examples

This example converts a binary quadratic model to NumPy array format while ordering variables and adding one ('d').

```
>>> import dimod
>>> import numpy as np
...
>>> model = dimod.BinaryQuadraticModel({'a': 1, 'b': -1, 'c': .5},
...                                    {('a', 'b'): .5, ('b', 'c'): 1.5},
...                                    1.4,
...                                    dimod.BINARY)
>>> model.to_numpy_matrix(variable_order=['d', 'c', 'b', 'a'])
array([[ 0. ,  0. ,  0. ,  0. ],
       [ 0. ,  0.5,  1.5,  0. ],
       [ 0. ,  0. , -1. ,  0.5],
       [ 0. ,  0. ,  0. ,  1. ]])
```

dimod.BinaryQuadraticModel.to_numpy_vectors

BinaryQuadraticModel.**to_numpy_vectors**(*variable_order=None, dtype=<class 'float'>, index_dtype=<class 'numpy.int64'>, sort_indices=False*)

Convert a binary quadratic model to numpy arrays.

Parameters

- **variable_order** (*iterable, optional*) – If provided, labels the variables; otherwise, row/column indices are used.
- **dtype** (numpy.dtype, optional) – Data-type of the biases. By default, the data-type is inferred from the biases.
- **index_dtype** (numpy.dtype, optional) – Data-type of the indices. By default, the data-type is inferred from the labels.
- **sort_indices** (*bool, optional, default=False*) – If True, the indices are sorted, first by row then by column. Otherwise they match *quadratic*.

Returns

A numpy array of the linear biases.

tuple: The quadratic biases in COOrdinate format.

ndarray: A numpy array of the row indices of the quadratic matrix entries

ndarray: A numpy array of the column indices of the quadratic matrix entries

ndarray: A numpy array of the values of the quadratic matrix entries

The offset

Return type ndarray

Examples

```
>>> bqm = dimod.BinaryQuadraticModel({}, {(0, 1): .5, (3, 2): -1, (0, 3): 1.5}, 0.
↪0, dimod.SPIN)
>>> lin, (i, j, vals), off = bqm.to_numpy_vectors(sort_indices=True)
>>> lin
array([0., 0., 0., 0.])
>>> i
array([0, 0, 2])
>>> j
array([1, 3, 3])
>>> vals
array([ 0.5,  1.5, -1. ])
```

dimod.BinaryQuadraticModel.to_qubo

`BinaryQuadraticModel.to_qubo()`

Convert a binary quadratic model to QUBO format.

If the binary quadratic model's vartype is not `Vartype.BINARY`, values are converted.

Returns 2-tuple of form (*biases*, *offset*), where *biases* is a dict in which keys are pairs of variables and values are the associated linear or quadratic bias and *offset* is a number that represents the constant offset of the binary quadratic model.

Return type tuple

Examples

This example converts a binary quadratic model with spin variables to QUBO format with binary variables.

```
>>> import dimod
>>> model = dimod.BinaryQuadraticModel({0: 1, 1: -1, 2: .5},
...                                    {(0, 1): .5, (1, 2): 1.5},
...                                    1.4,
...                                    dimod.SPIN)
>>> model.to_qubo()    # doctest: +SKIP
({(0, 0): 1.0, (0, 1): 2.0, (1, 1): -6.0, (1, 2): 6.0, (2, 2): -2.0}, 2.9)
```

dimod.BinaryQuadraticModel.to_pandas_dataframe

`BinaryQuadraticModel.to_pandas_dataframe()`

Convert a binary quadratic model to pandas DataFrame format.

Returns The binary quadratic model as a DataFrame. The DataFrame has binary vartype. The rows and columns are labeled by the variables in the binary quadratic model.

Return type `pandas.DataFrame`

Notes

The DataFrame representation of a binary quadratic model only makes sense for binary models. For a binary sample x, the energy of the model is given by:

$$E(x) = x^T Q x$$

The offset is dropped when converting to a pandas DataFrame.

Examples

This example converts a binary quadratic model to pandas DataFrame format.

```
>>> import dimod
>>> model = dimod.BinaryQuadraticModel({'a': 1.1, 'b': -1., 'c': .5},
...                                    {('a', 'b'): .5, ('b', 'c'): 1.5},
...                                    1.4,
...                                    dimod.BINARY)
>>> model.to_pandas_dataframe()   # doctest: +SKIP
     a    b    c
a  1.1  0.5  0.0
b  0.0 -1.0  1.5
c  0.0  0.0  0.5
```

dimod.BinaryQuadraticModel.to_serializable

BinaryQuadraticModel.**to_serializable**(*use_bytes=False, bias_dtype=<class 'numpy.float32'>, bytes_type=<class 'bytes'>*)

Convert the binary quadratic model to a serializable object.

Parameters

- **use_bytes** (*bool, optional, default=False*) – If True, a compact representation representing the biases as bytes is used. Uses `tobytes()`.
- **bias_dtype** (*data-type, optional, default=numpy.float32*) – If *use_bytes* is True, this `dtype` will be used to represent the bias values in the serialized format.
- **bytes_type** (*class, optional, default=bytes*) – This class will be used to wrap the bytes objects in the serialization if *use_bytes* is true. Useful for when using Python 2 and using BSON encoding, which will not accept the raw *bytes* type, so *bson.Binary* can be used instead.

Returns An object that can be serialized.

Return type dict

Examples

Encode using JSON

```
>>> import dimod
>>> import json
...
```

(continues on next page)

(continued from previous page)

```
>>> bqm = dimod.BinaryQuadraticModel({'a': -1.0, 'b': 1.0}, {('a', 'b'): -1.0}, 0.
→0, dimod.SPIN)
>>> s = json.dumps(bqm.to_serializable())
```

Encode using BSON in python 3.5+

```
>>> import dimod
>>> import bson
...
>>> bqm = dimod.BinaryQuadraticModel({'a': -1.0, 'b': 1.0}, {('a', 'b'): -1.0}, 0.
→0, dimod.SPIN)
>>> doc = bqm.to_serializable(use_bytes=True)
>>> b = bson.BSON.encode(doc)   # doctest: +SKIP
```

Encode using BSON in python 2.7. Because `bytes` is an alias for `str`, we need to signal to the encoder that it should encode the biases and labels as binary data.

```
>>> import dimod
>>> import bson
...
>>> bqm = dimod.BinaryQuadraticModel({'a': -1.0, 'b': 1.0}, {('a', 'b'): -1.0}, 0.
→0, dimod.SPIN)
>>> doc = bqm.to_serializable(use_bytes=True, bytes_type=bson.Binary)
>>> b = bson.BSON.encode(doc)   # doctest: +SKIP
```

See also:

`from_serializable()`

`json.dumps()`, `json.dump()` JSON encoding functions

`bson.BSON.encode()` BSON encoding method

Alias

`BQM`

Alias for *`BinaryQuadraticModel`*

alias of `dimod.binary_quadratic_model.BinaryQuadraticModel`

BQM Generators

Chimera Structured

`chimera_anticluster`(m[, n, t, multiplier, . . .])	Generate an anticluster problem on a Chimera lattice.

dimod.generators.chimera.chimera_anticluster

`chimera_anticluster`(*m, n=None, t=4, multiplier=3.0, cls=<class 'dimod.binary_quadratic_model.BinaryQuadraticModel'>, subgraph=None, seed=None*)

Generate an anticluster problem on a Chimera lattice.

An anticluster problem has weak interactions within a tile and strong interactions between tiles.

Parameters

- **m** (*int*) – Number of rows in the Chimera lattice.
- **n** (*int, optional, default=m*) – Number of columns in the Chimera lattice.
- **t** (*int, optional, default=t*) – Size of the shore within each Chimera tile.
- **multiplier** (*number, optional, default=3.0*) – Strength of the intertile edges.
- **cls** (class, optional, default=:class:.*BinaryQuadraticModel*) – Binary quadratic model class to build from.
- **subgraph** (int/tuple[nodes, edges]/list[edge]/`Graph`) – A subgraph of a Chimera(m, n, t) graph to build the anticluster problem on.
- **seed** (*int, optional, default=None*) – Random seed.

Returns spin-valued binary quadratic model.

Return type *BinaryQuadraticModel*

Constraints

combinations(n, k[, strength, vartype])	Generate a bqm that is minimized when k of n variables are selected.

dimod.generators.constraints.combinations

combinations (*n*, *k*, *strength=1*, *vartype=<Vartype.BINARY: frozenset({0, 1})>*)

Generate a bqm that is minimized when k of n variables are selected.

More fully, we wish to generate a binary quadratic model which is minimized for each of the k-combinations of its variables.

The energy for the binary quadratic model is given by $(\sum_i x_i - k)^2$.

Parameters

- **n** (*int/list/set*) – If n is an integer, variables are labelled [0, n-1]. If n is list or set then the variables are labelled accordingly.
- **k** (*int*) – The generated binary quadratic model will have 0 energy when any k of the variables are 1.
- **strength** (*number, optional, default=1*) – The energy of the first excited state of the binary quadratic model.
- **vartype** (*Vartype*/str/set) – Variable type for the binary quadratic model. Accepted input values:
  - `Vartype.SPIN`, `'SPIN'`, `{-1, 1}`
  - `Vartype.BINARY`, `'BINARY'`, `{0, 1}`

Returns *BinaryQuadraticModel*

Examples

```
>>> bqm = dimod.generators.combinations(['a', 'b', 'c'], 2)
>>> bqm.energy({'a': 1, 'b': 0, 'c': 1})
0.0
>>> bqm.energy({'a': 1, 'b': 1, 'c': 1})
1.0
```

```
>>> bqm = dimod.generators.combinations(5, 1)
>>> bqm.energy({0: 0, 1: 0, 2: 1, 3: 0, 4: 0})
0.0
>>> bqm.energy({0: 0, 1: 0, 2: 1, 3: 1, 4: 0})
1.0
```

```
>>> bqm = dimod.generators.combinations(['a', 'b', 'c'], 2, strength=3.0)
>>> bqm.energy({'a': 1, 'b': 0, 'c': 1})
0.0
>>> bqm.energy({'a': 1, 'b': 1, 'c': 1})
3.0
```

Frustrated Cluster Loops

`frustrated_loop`(graph, num_cycles[, R, . . .])	Generate a frustrated loop problem.

dimod.generators.fcl.frustrated_loop

frustrated_loop (*graph*, *num_cycles*, *R=inf*, *cycle_predicates=()*, *max_failed_cycles=100*, *seed=None*)

Generate a frustrated loop problem.

A (generic) frustrated loop (FL) problem is a sum of Hamiltonians, each generated from a single "good" loop. 1. Generate a loop by random walking on the support graph. 2. If the cycle is "good" (according to provided predicates), continue, else go to 1. 3. Choose one edge of the loop to be anti-ferromagnetic; all other edges are ferromagnetic. 4. Add the loop's coupler values to the FL problem. If at any time the magnitude of a coupler in the FL problem exceeds a given precision *R*, remove that coupler from consideration in the loop generation procedure.

This is a generic generator of FL problems that encompasses both the original FL problem definition from[1] and the limited FL problem definition from[2]

Parameters

- **graph** (int/tuple[nodes, edges]/list[edge]/`Graph`) – The graph to build the frustrated loops on. Either an integer n, interpreted as a complete graph of size n, a nodes/edges pair, a list of edges or a NetworkX graph.
- **num_cyles** (*int*) – Desired number of frustrated cycles.
- **R** (*int, optional, default=inf*) – Maximum interaction weight.
- **cycle_predicates** (*tuple[function], optional*) – An iterable of functions, which should accept a cycle and return a bool.

[1] Hen, I., J. Job, T. Albash, T.F. Rønnow, M. Troyer, D. Lidar. Probing for quantum speedup in spin glass problems with planted solutions. https://arxiv.org/abs/1502.01663v2

[2] King, A.D., T. Lanting, R. Harris. Performance of a quantum annealer on range-limited constraint satisfaction problems. https://arxiv.org/abs/1502.02098

- **max_failed_cycles** (*int, optional, default=100*) – Maximum number of failures to find a cycle before terminating.
- **seed** (*int, optional, default=None*) – Random seed.

Random

randint(graph, vartype[, low, high, cls, seed])	Generate a bqm with random biases and offset.
ran_r(r, graph[, cls, seed])	Generate an Ising model for a RANr problem.
uniform(graph, vartype[, low, high, cls, seed])	Generate a bqm with random biases and offset.

dimod.generators.random.randint

randint (*graph*, *vartype*, *low=0*, *high=1*, *cls=<class 'dimod.binary_quadratic_model.BinaryQuadraticModel'>*, *seed=None*)

Generate a bqm with random biases and offset.

Biases and offset are integer-valued in range [low, high] inclusive.

Parameters

- **graph** (int/tuple[nodes, edges]/list[edge]/Graph) – The graph to build the bqm on. Either an integer n, interpreted as a complete graph of size n, a nodes/edges pair, a list of edges or a NetworkX graph.
- **vartype** (*Vartype*/str/set) – Variable type for the binary quadratic model. Accepted input values:
  - `Vartype.SPIN`, `'SPIN'`, `{-1, 1}`
  - `Vartype.BINARY`, `'BINARY'`, `{0, 1}`
- **low** (*float, optional, default=0*) – The low end of the range for the random biases.
- **high** (*float, optional, default=1*) – The high end of the range for the random biases.
- **cls** (*BinaryQuadraticModel*) – Binary quadratic model class to build from.
- **seed** (*int, optional, default=None*) – Random seed.

Returns *BinaryQuadraticModel*

dimod.generators.random.ran_r

ran_r (*r*, *graph*, *cls=<class 'dimod.binary_quadratic_model.BinaryQuadraticModel'>*, *seed=None*)

Generate an Ising model for a RANr problem.

In RANr problems all linear biases are zero and quadratic values are uniformly selected integers between -r to r, excluding zero. This class of problems is relevant for binary quadratic models (BQM) with spin variables (Ising models).

This generator of RANr problems follows the definition in [Kin2015].

Parameters

- **r** (*int*) – Order of the RANr problem.

- **graph** (int/tuple[nodes, edges]/list[edge]/Graph) – The graph to build the bqm on. Either an integer n, interpreted as a complete graph of size n, a nodes/edges pair, a list of edges or a NetworkX graph.
- **cls** (*BinaryQuadraticModel*) – Binary quadratic model class to build from.
- **seed** (*int, optional, default=None*) – Random seed.

Returns *BinaryQuadraticModel*.

Examples:

```
>>> import networkx as nx
>>> K_7 = nx.complete_graph(7)
>>> bqm = dimod.generators.random.ran_r(1, K_7)
```

dimod.generators.random.uniform

uniform(*graph, vartype, low=0.0, high=1.0, cls=<class 'dimod.binary_quadratic_model.BinaryQuadraticModel'>, seed=None*)

Generate a bqm with random biases and offset.

Biases and offset are drawn from a uniform distribution range (low, high).

Parameters

- **graph** (int/tuple[nodes, edges]/list[edge]/Graph) – The graph to build the bqm on. Either an integer n, interpreted as a complete graph of size n, a nodes/edges pair, a list of edges or a NetworkX graph.
- **vartype** (*Vartype*/str/set) – Variable type for the binary quadratic model. Accepted input values:
  - Vartype.SPIN, 'SPIN', {-1, 1}
  - Vartype.BINARY, 'BINARY', {0, 1}
- **low** (*float, optional, default=0.0*) – The low end of the range for the random biases.
- **high** (*float, optional, default=1.0*) – The high end of the range for the random biases.
- **cls** (*BinaryQuadraticModel*) – Binary quadratic model class to build from.
- **seed** (*int, optional, default=None*) – Random seed.

Returns *BinaryQuadraticModel*

BQM Functions

Functional interface to BQM methods and assorted utilities.

Roof-duality

fix_variables(bqm[, sampling_mode])	Determine assignments for some variables of a binary quadratic model.

dimod.roof_duality.fix_variables

fix_variables(*bqm*, *sampling_mode=True*)

Determine assignments for some variables of a binary quadratic model.

Roof duality finds a lower bound for the minimum of a quadratic polynomial. It can also find minimizing assignments for some of the polynomial's variables; these fixed variables take the same values in all optimal solutions [BHT] [BH]. A quadratic pseudo-Boolean function can be represented as a network to find the lower bound through network-flow computations. *fix_variables* uses maximum flow in the implication network to correctly fix variables. Consequently, you can find an assignment for the remaining variables that attains the optimal value.

Parameters

- **bqm** (*BinaryQuadraticModel*) – A binary quadratic model.
- **sampling_mode** (*bool, optional, default=True*) – In sampling mode, only roof-duality is used. When *sampling_mode* is false, strongly connected components are used to fix more variables, but in some optimal solutions these variables may take different values.

Returns Variable assignments for some variables of the specified binary quadratic model.

Return type dict

Examples

This example creates a binary quadratic model with a single ground state and fixes the model's single variable to the minimizing assignment.

```
>>> bqm = dimod.BinaryQuadraticModel.empty(dimod.SPIN)
>>> bqm.add_variable('a', 1.0)
>>> dimod.fix_variables(bqm)
{'a': -1}
```

This example has two ground states, $a = b = -1$ and $a = b = 1$, with no variable having a single value for all ground states, so neither variable is fixed.

```
>>> bqm = dimod.BinaryQuadraticModel.empty(dimod.SPIN)
>>> bqm.add_interaction('a', 'b', -1.0)
>>> dimod.fix_variables(bqm) # doctest: +SKIP
{}
```

This example turns sampling model off, so variables are fixed to an assignment that attains the ground state.

```
>>> bqm = dimod.BinaryQuadraticModel.empty(dimod.SPIN)
>>> bqm.add_interaction('a', 'b', -1.0)
>>> dimod.fix_variables(bqm, sampling_mode=False) # doctest: +SKIP
{'a': 1, 'b': 1}
```

Traversal

connected_components(bqm)	Yields sets of connected variables.
bfs_variables(bqm, source)	Yields variables in breadth-first search order.

dimod.traversal.connected_components

connected_components (*bqm*)
Yields sets of connected variables.

Parameters **bqm** (*dimod.BinaryQuadraticModel*) – A binary quadratic model.

Yields *set* – A set of variables in the bqm that form a connected component.

dimod.traversal.bfs_variables

bfs_variables (*bqm, source*)
Yields variables in breadth-first search order.

Parameters

- **bqm** (*dimod.BinaryQuadraticModel*) – A binary quadratic model.
- **source** (*variable*) – A variable in the binary quadratic model.

Yields *variable* – variables in the bqm, yielded in breadth-first search order starting at source.

2.2.2 Samplers and Composites

Samplers

The *dimod* package includes several example samplers.

Contents

Exact Solver

A solver that calculates the energy of all possible samples.

Note: This sampler is designed for use in testing. Because it calculates the energy for every possible sample, it is very slow.

Class

class ExactSolver

A simple exact solver for testing and debugging code using your local CPU.

Notes

This solver becomes slow for problems with 18 or more variables.

Examples

This example solves a two-variable Ising model.

```
>>> h = {'a': -0.5, 'b': 1.0}
>>> J = {('a', 'b'): -1.5}
>>> sampleset = dimod.ExactSolver().sample_ising(h, J)
>>> print(sampleset)
   a  b energy num_oc.
0 -1 -1   -2.0       1
2 +1 +1   -1.0       1
1 +1 -1    0.0       1
3 -1 +1    3.0       1
['SPIN', 4 rows, 4 samples, 2 variables]
```

This example solves a two-variable QUBO.

```
>>> Q = {('a', 'b'): 2.0, ('a', 'a'): 1.0, ('b', 'b'): -0.5}
>>> sampleset = dimod.ExactSolver().sample_qubo(Q)
```

This example solves a two-variable binary quadratic model

```
>>> bqm = dimod.BinaryQuadraticModel({'a': 1.5}, {('a', 'b'): -1}, 0.0, 'SPIN')
>>> sampleset = dimod.ExactSolver().sample(bqm)
```

Methods

ExactSolver.sample(bqm)	Sample from a binary quadratic model.
ExactSolver.sample_ising(h, J, **parameters)	Sample from an Ising model using the implemented sample method.
ExactSolver.sample_qubo(Q, **parameters)	Sample from a QUBO using the implemented sample method.

dimod.reference.samplers.exact_solver.ExactSolver.sample

ExactSolver.**sample**(*bqm*)

Sample from a binary quadratic model.

Parameters **bqm** (*BinaryQuadraticModel*) – Binary quadratic model to be sampled from.

Returns *SampleSet*

dimod.reference.samplers.exact_solver.ExactSolver.sample_ising

ExactSolver.**sample_ising**(*h*, *J*, ***parameters*)

Sample from an Ising model using the implemented sample method.

This method is inherited from the *Sampler* base class.

Converts the Ising model into a *BinaryQuadraticModel* and then calls *sample()*.

Parameters

- **h** (*dict/list*) – Linear biases of the Ising problem. If a dict, should be of the form *{v: bias, …}* where is a spin-valued variable and *bias* is its associated bias. If a list, it is treated as a list of biases where the indices are the variable labels.
- **J** (*dict[(variable, variable), bias]*) – Quadratic biases of the Ising problem.
- ****kwargs** – See the implemented sampling for additional keyword definitions.

Returns *SampleSet*

See also:

sample(), *sample_qubo()*

dimod.reference.samplers.exact_solver.ExactSolver.sample_qubo

ExactSolver.**sample_qubo**(*Q*, ***parameters*)

Sample from a QUBO using the implemented sample method.

This method is inherited from the *Sampler* base class.

Converts the QUBO into a *BinaryQuadraticModel* and then calls *sample()*.

Parameters

- **Q** (*dict*) – Coefficients of a quadratic unconstrained binary optimization (QUBO) problem. Should be a dict of the form *{(u, v): bias, …}* where *u*, *v*, are binary-valued variables and *bias* is their associated coefficient.
- ****kwargs** – See the implemented sampling for additional keyword definitions.

Returns *SampleSet*

See also:

sample(), *sample_ising()*

Null Sampler

A sampler that always returns an empty sample set.

Class

class NullSampler(*parameters=None*)

A sampler that always returns an empty sample set.

This sampler is useful for writing unit tests where the result is not important.

Parameters parameters (*iterable/dict, optional*) – If provided, sets the parameters accepted by the sample methods. The values given in these parameters are ignored.

Examples

```
>>> bqm = dimod.BinaryQuadraticModel.from_qubo({('a', 'b'): 1})
>>> sampler = dimod.NullSampler()
>>> sampleset = sampler.sample(bqm)
>>> len(sampleset)
0
```

Setting additional parameters for the null sampler.

```
>>> bqm = dimod.BinaryQuadraticModel.from_qubo({('a', 'b'): 1})
>>> sampler = dimod.NullSampler(parameters=['a'])
>>> sampleset = sampler.sample(bqm, a=5)
```

Properties

NullSampler.parameters	Keyword arguments accepted by the sampling methods

dimod.reference.samplers.null_sampler.NullSampler.parameters

NullSampler.**parameters = None**

Keyword arguments accepted by the sampling methods

Methods

NullSampler.sample(bqm, **kwargs)	Return an empty sample set.
NullSampler.sample_ising(h, J, **parameters)	Sample from an Ising model using the implemented sample method.
NullSampler.sample_qubo(Q, **parameters)	Sample from a QUBO using the implemented sample method.

dimod.reference.samplers.null_sampler.NullSampler.sample

NullSampler.**sample**(*bqm*, ***kwargs*)

Return an empty sample set.

Parameters

- **bqm** (*BinaryQuadraticModel*) – The binary quadratic model determines the variables labels in the sample set.
- **kwargs** – As specified when constructing the null sampler.

Returns The empty sample set.

Return type *SampleSet*

dimod.reference.samplers.null_sampler.NullSampler.sample_ising

NullSampler.**sample_ising**(*h*, *J*, ***parameters*)

Sample from an Ising model using the implemented sample method.

This method is inherited from the *Sampler* base class.

Converts the Ising model into a *BinaryQuadraticModel* and then calls *sample()*.

Parameters

- **h** (*dict/list*) – Linear biases of the Ising problem. If a dict, should be of the form *{v: bias, . . . }* where is a spin-valued variable and *bias* is its associated bias. If a list, it is treated as a list of biases where the indices are the variable labels.
- **J** (*dict[(variable, variable), bias]*) – Quadratic biases of the Ising problem.
- ****kwargs** – See the implemented sampling for additional keyword definitions.

Returns *SampleSet*

See also:

sample(), *sample_qubo()*

dimod.reference.samplers.null_sampler.NullSampler.sample_qubo

NullSampler.**sample_qubo**(*Q*, ***parameters*)

Sample from a QUBO using the implemented sample method.

This method is inherited from the *Sampler* base class.

Converts the QUBO into a *BinaryQuadraticModel* and then calls *sample()*.

Parameters

- **Q** (*dict*) – Coefficients of a quadratic unconstrained binary optimization (QUBO) problem. Should be a dict of the form *{(u, v): bias, . . . }* where *u*, *v*, are binary-valued variables and *bias* is their associated coefficient.
- ****kwargs** – See the implemented sampling for additional keyword definitions.

Returns *SampleSet*

See also:

sample(), *sample_ising()*

Random Sampler

A sampler that gives random samples.

Class

class RandomSampler
A sampler that gives random samples for testing.

Properties

RandomSampler.parameters	Keyword arguments accepted by the sampling methods.

dimod.reference.samplers.random_sampler.RandomSampler.parameters

RandomSampler.**parameters = None**
Keyword arguments accepted by the sampling methods.

Contents are exactly *{'num_reads': []}*

Type dict

Methods

RandomSampler.sample(bqm[, num_reads])	Give random samples for a binary quadratic model.
RandomSampler.sample_ising(h, J, **parameters)	Sample from an Ising model using the implemented sample method.
RandomSampler.sample_qubo(Q, **parameters)	Sample from a QUBO using the implemented sample method.

dimod.reference.samplers.random_sampler.RandomSampler.sample

RandomSampler.**sample**(*bqm*, *num_reads=10*)
Give random samples for a binary quadratic model.

Variable assignments are chosen by coin flip.

Parameters

- **bqm** (*BinaryQuadraticModel*) – Binary quadratic model to be sampled from.
- **num_reads** (*int, optional, default=10*) – Number of reads.

Returns *SampleSet*

dimod.reference.samplers.random_sampler.RandomSampler.sample_ising

RandomSampler.**sample_ising**(*h*, *J*, ***parameters*)

Sample from an Ising model using the implemented sample method.

This method is inherited from the `Sampler` base class.

Converts the Ising model into a `BinaryQuadraticModel` and then calls `sample()`.

Parameters

- **h** (*dict/list*) – Linear biases of the Ising problem. If a dict, should be of the form *{v: bias, …}* where is a spin-valued variable and *bias* is its associated bias. If a list, it is treated as a list of biases where the indices are the variable labels.
- **J** (*dict[(variable, variable), bias]*) – Quadratic biases of the Ising problem.
- ****kwargs** – See the implemented sampling for additional keyword definitions.

Returns `SampleSet`

See also:

`sample()`, `sample_qubo()`

dimod.reference.samplers.random_sampler.RandomSampler.sample_qubo

RandomSampler.**sample_qubo**(*Q*, ***parameters*)

Sample from a QUBO using the implemented sample method.

This method is inherited from the `Sampler` base class.

Converts the QUBO into a `BinaryQuadraticModel` and then calls `sample()`.

Parameters

- **Q** (*dict*) – Coefficients of a quadratic unconstrained binary optimization (QUBO) problem. Should be a dict of the form *{(u, v): bias, …}* where *u*, *v*, are binary-valued variables and *bias* is their associated coefficient.
- ****kwargs** – See the implemented sampling for additional keyword definitions.

Returns `SampleSet`

See also:

`sample()`, `sample_ising()`

Simulated Annealing Sampler

A reference implementation of a simulated annealing sampler.

`neal.sampler.SimulatedAnnealingSampler` is a more performant implementation of simulated annealing you can use for solving problems.

Class

class SimulatedAnnealingSampler

A simple simulated annealing sampler for testing and debugging code.

Examples

This example solves a two-variable Ising model.

```
>>> h = {'a': -0.5, 'b': 1.0}
>>> J = {('a', 'b'): -1.5}
>>> sampleset = dimod.SimulatedAnnealingSampler().sample_ising(h, J)
```

Properties

SimulatedAnnealingSampler.parameters	Keyword arguments accepted by the sampling methods.

dimod.reference.samplers.simulated_annealing.SimulatedAnnealingSampler.parameters

SimulatedAnnealingSampler.**parameters = None**

Keyword arguments accepted by the sampling methods.

Contents are exactly *{'beta_range': [], num_reads': [], 'num_sweeps': []}*

Type dict

Methods

SimulatedAnnealingSampler.sample(bqm[, ...])	Sample from low-energy spin states using simulated annealing.
SimulatedAnnealingSampler.sample_ising(h, J, ...)	Sample from an Ising model using the implemented sample method.
SimulatedAnnealingSampler.sample_qubo(Q, ...)	Sample from a QUBO using the implemented sample method.

dimod.reference.samplers.simulated_annealing.SimulatedAnnealingSampler.sample

SimulatedAnnealingSampler.**sample**(*bqm*, *beta_range=None*, *num_reads=10*, *num_sweeps=1000*)

Sample from low-energy spin states using simulated annealing.

Parameters

- **bqm** (*BinaryQuadraticModel*) – Binary quadratic model to be sampled from.
- **beta_range** (*tuple, optional*) – Beginning and end of the beta schedule (beta is the inverse temperature) as a 2-tuple. The schedule is applied linearly in beta. Default is chosen based on the total bias associated with each node.
- **num_reads** (*int, optional, default=10*) – Number of reads. Each sample is the result of a single run of the simulated annealing algorithm.
- **num_sweeps** (*int, optional, default=1000*) – Number of sweeps or steps.

Returns *SampleSet*

Note: This is a reference implementation, not optimized for speed and therefore not an appropriate sampler for benchmarking.

dimod.reference.samplers.simulated_annealing.SimulatedAnnealingSampler.sample_ising

SimulatedAnnealingSampler.**sample_ising**(*h*, *J*, ***parameters*)

Sample from an Ising model using the implemented sample method.

This method is inherited from the *Sampler* base class.

Converts the Ising model into a *BinaryQuadraticModel* and then calls *sample()*.

Parameters

- **h** (*dict/list*) – Linear biases of the Ising problem. If a dict, should be of the form *{v: bias, . . . }* where is a spin-valued variable and *bias* is its associated bias. If a list, it is treated as a list of biases where the indices are the variable labels.
- **J** (*dict[(variable, variable), bias]*) – Quadratic biases of the Ising problem.
- ****kwargs** – See the implemented sampling for additional keyword definitions.

Returns *SampleSet*

See also:

sample(), *sample_qubo()*

dimod.reference.samplers.simulated_annealing.SimulatedAnnealingSampler.sample_qubo

SimulatedAnnealingSampler.**sample_qubo**(*Q*, ***parameters*)

Sample from a QUBO using the implemented sample method.

This method is inherited from the *Sampler* base class.

Converts the QUBO into a *BinaryQuadraticModel* and then calls *sample()*.

Parameters

- **Q** (*dict*) – Coefficients of a quadratic unconstrained binary optimization (QUBO) problem. Should be a dict of the form *{(u, v): bias, . . . }* where *u*, *v*, are binary-valued variables and *bias* is their associated coefficient.
- ****kwargs** – See the implemented sampling for additional keyword definitions.

Returns *SampleSet*

See also:

sample(), *sample_ising()*

Composites

The *dimod* package includes several example composed samplers.

Connected Components Composite

A composite that breaks the problem into sub-problems corresponding to the connected components of the binary quadratic model graph before sending to its child sampler.

Class

class ConnectedComponentsComposite(*child_sampler*)
Composite to decompose a problem to the connected components and solve each.

Connected components of a bqm graph are computed (if not provided), and each subproblem is passed to the child sampler. Returned samples from each child sampler are merged. Only the best solution of each response is pick and merge with others (i.e. this composite returns a single solution).

Parameters **sampler** (*dimod.Sampler*) – A dimod sampler

Examples

This example uses *ConnectedComponentsComposite* to instantiate a composed sampler that submits a simple Ising problem to a sampler. The composed sampler finds the connected components and solves each.

```
>>> h = {1: -1.3, 2: 2.3, 3:-1.2, 4: -0.5}
>>> J = {(1, 4): -0.6, (1, 3): 0.6, (3, 4): 1.0, (2, 3): -1.0}
>>> sampler = dimod.ConnectedComponentsComposite(dimod.ExactSolver())
>>> sampleset = sampler.sample_ising(h, J)
```

Properties

ConnectedComponentsComposite.child	The child sampler.
ConnectedComponentsComposite.children	List of child samplers that that are used by this composite.
ConnectedComponentsComposite.parameters	A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevent to each parameter.
ConnectedComponentsComposite.properties	A dict containing any additional information about the sampler.

dimod.reference.composites.connectedcomponent.ConnectedComponentsComposite.child

ConnectedComponentsComposite.**child**
The child sampler. First sampler in `Composite.children`.

Type *Sampler*

dimod.reference.composites.connectedcomponent.ConnectedComponentsComposite.children

ConnectedComponentsComposite.**children**
List of child samplers that that are used by this composite.

Type list[*Sampler*]

dimod.reference.composites.connectedcomponent.ConnectedComponentsComposite.parameters

ConnectedComponentsComposite.**parameters**

A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevent to each parameter.

Type dict

dimod.reference.composites.connectedcomponent.ConnectedComponentsComposite.properties

ConnectedComponentsComposite.**properties**

A dict containing any additional information about the sampler.

Type dict

Methods

ConnectedComponentsComposite.sample(bqm[, ...])	Sample from the provided binary quadratic model.
ConnectedComponentsComposite.sample_ising(h, ...)	Sample from an Ising model using the implemented sample method.
ConnectedComponentsComposite.sample_qubo(Q, ...)	Sample from a QUBO using the implemented sample method.

dimod.reference.composites.connectedcomponent.ConnectedComponentsComposite.sample

ConnectedComponentsComposite.**sample**(*bqm*, *components=None*, ***parameters*)

Sample from the provided binary quadratic model.

Parameters

- **bqm** (*dimod.BinaryQuadraticModel*) – Binary quadratic model to be sampled from.
- **components** (*list(set)*) – A list of disjoint set of variables that fully partition the variables
- ****parameters** – Parameters for the sampling method, specified by the child sampler.

Returns *dimod.SampleSet*

dimod.reference.composites.connectedcomponent.ConnectedComponentsComposite.sample_ising

ConnectedComponentsComposite.**sample_ising**(*h*, *J*, ***parameters*)

Sample from an Ising model using the implemented sample method.

This method is inherited from the *Sampler* base class.

Converts the Ising model into a *BinaryQuadraticModel* and then calls *sample()*.

Parameters

- **h** (*dict/list*) – Linear biases of the Ising problem. If a dict, should be of the form *{v: bias, ...}* where is a spin-valued variable and *bias* is its associated bias. If a list, it is treated as a list of biases where the indices are the variable labels.

- **J** (*dict[(variable, variable), bias]*) – Quadratic biases of the Ising problem.
- ****kwargs** – See the implemented sampling for additional keyword definitions.

Returns *SampleSet*

See also:

sample(), *sample_qubo()*

dimod.reference.composites.connectedcomponent.ConnectedComponentsComposite.sample_qubo

ConnectedComponentsComposite.**sample_qubo**(*Q*, ***parameters*)

Sample from a QUBO using the implemented sample method.

This method is inherited from the *Sampler* base class.

Converts the QUBO into a *BinaryQuadraticModel* and then calls *sample()*.

Parameters

- **Q** (*dict*) – Coefficients of a quadratic unconstrained binary optimization (QUBO) problem. Should be a dict of the form *{(u, v): bias, …}* where *u*, *v*, are binary-valued variables and *bias* is their associated coefficient.
- ****kwargs** – See the implemented sampling for additional keyword definitions.

Returns *SampleSet*

See also:

sample(), *sample_ising()*

Clip Composite

A composite that clips problem variables below and above threshold. if lower and upper bounds is not given it does nothing.

Class

class ClipComposite(*child_sampler*)

Composite to clip variables of a problem

Clips the variables of a bqm and modifies linear and quadratic terms accordingly.

Parameters **sampler** (*dimod.Sampler*) – A dimod sampler

Examples

This example uses *ClipComposite* to instantiate a composed sampler that submits a simple Ising problem to a sampler. The composed sampler clips linear and quadratic biases as indicated by options.

```
>>> h = {'a': -4.0, 'b': -4.0}
>>> J = {('a', 'b'): 3.2}
>>> sampler = dimod.ClipComposite(dimod.ExactSolver())
>>> response = sampler.sample_ising(h, J, lower_bound=-2.0, upper_bound=2.0)
```

Properties

ClipComposite.child	The child sampler.
ClipComposite.children	List of child samplers that that are used by this composite.
ClipComposite.parameters	A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevent to each parameter.
ClipComposite.properties	A dict containing any additional information about the sampler.

dimod.reference.composites.clipcomposite.ClipComposite.child

ClipComposite.**child**

The child sampler. First sampler in `Composite.children`.

Type *Sampler*

dimod.reference.composites.clipcomposite.ClipComposite.children

ClipComposite.**children**

List of child samplers that that are used by this composite.

Type list[*Sampler*]

dimod.reference.composites.clipcomposite.ClipComposite.parameters

ClipComposite.**parameters**

A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevent to each parameter.

Type dict

dimod.reference.composites.clipcomposite.ClipComposite.properties

ClipComposite.**properties**

A dict containing any additional information about the sampler.

Type dict

Methods

ClipComposite.sample(bqm[, lower_bound, ...])	Clip and sample from the provided binary quadratic model.
ClipComposite.sample_ising(h, J, **parameters)	Sample from an Ising model using the implemented sample method.
ClipComposite.sample_qubo(Q, **parameters)	Sample from a QUBO using the implemented sample method.

dimod.reference.composites.clipcomposite.ClipComposite.sample

ClipComposite.**sample**(*bqm*, *lower_bound=None*, *upper_bound=None*, ***parameters*)

Clip and sample from the provided binary quadratic model.

If lower_bound and upper_bound are given variables with value above or below are clipped.

Parameters

- **bqm** (*dimod.BinaryQuadraticModel*) – Binary quadratic model to be sampled from.
- **lower_bound** (*number*) – Value by which to clip the variables from below.
- **upper_bound** (*number*) – Value by which to clip the variables from above.
- ****parameters** – Parameters for the sampling method, specified by the child sampler.

Returns *dimod.SampleSet*

dimod.reference.composites.clipcomposite.ClipComposite.sample_ising

ClipComposite.**sample_ising**(*h*, *J*, ***parameters*)

Sample from an Ising model using the implemented sample method.

This method is inherited from the *Sampler* base class.

Converts the Ising model into a *BinaryQuadraticModel* and then calls *sample()*.

Parameters

- **h** (*dict/list*) – Linear biases of the Ising problem. If a dict, should be of the form *{v: bias, . . . }* where is a spin-valued variable and *bias* is its associated bias. If a list, it is treated as a list of biases where the indices are the variable labels.
- **J** (*dict[(variable, variable), bias]*) – Quadratic biases of the Ising problem.
- ****kwargs** – See the implemented sampling for additional keyword definitions.

Returns *SampleSet*

See also:

sample(), *sample_qubo()*

dimod.reference.composites.clipcomposite.ClipComposite.sample_qubo

ClipComposite.**sample_qubo**(*Q*, ***parameters*)

Sample from a QUBO using the implemented sample method.

This method is inherited from the *Sampler* base class.

Converts the QUBO into a *BinaryQuadraticModel* and then calls *sample()*.

Parameters

- **Q** (*dict*) – Coefficients of a quadratic unconstrained binary optimization (QUBO) problem. Should be a dict of the form *{(u, v): bias, . . . }* where *u*, *v*, are binary-valued variables and *bias* is their associated coefficient.
- ****kwargs** – See the implemented sampling for additional keyword definitions.

Returns *SampleSet*

See also:

sample(), *sample_ising()*

Fixed Variable Composite

A composite that fixes the variables provided and removes them from the binary quadratic model before sending to its child sampler.

Class

class FixedVariableComposite(*child_sampler*)

Composite to fix variables of a problem to provided.

Fixes variables of a bqm and modifies linear and quadratic terms accordingly. Returned samples include the fixed variable

Parameters **sampler** (*dimod.Sampler*) – A dimod sampler

Examples

This example uses *FixedVariableComposite* to instantiate a composed sampler that submits a simple Ising problem to a sampler. The composed sampler fixes a variable and modifies linear and quadratic biases according.

```
>>> h = {1: -1.3, 4: -0.5}
>>> J = {(1, 4): -0.6}
>>> sampler = dimod.FixedVariableComposite(dimod.ExactSolver())
>>> sampleset = sampler.sample_ising(h, J, fixed_variables={1: -1})
```

Properties

FixedVariableComposite.child	The child sampler.
FixedVariableComposite.children	List of child samplers that that are used by this composite.
FixedVariableComposite.parameters	A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevent to each parameter.
FixedVariableComposite.properties	A dict containing any additional information about the sampler.

dimod.reference.composites.fixedvariable.FixedVariableComposite.child

FixedVariableComposite.**child**

The child sampler. First sampler in `Composite.children`.

Type *Sampler*

dimod.reference.composites.fixedvariable.FixedVariableComposite.children

FixedVariableComposite.**children**
List of child samplers that that are used by this composite.

Type list[*Sampler*]

dimod.reference.composites.fixedvariable.FixedVariableComposite.parameters

FixedVariableComposite.**parameters**
A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevent to each parameter.

Type dict

dimod.reference.composites.fixedvariable.FixedVariableComposite.properties

FixedVariableComposite.**properties**
A dict containing any additional information about the sampler.

Type dict

Methods

FixedVariableComposite.sample(bqm[, . . .])	Sample from the provided binary quadratic model.
FixedVariableComposite.sample_ising(h, J, . . .)	Sample from an Ising model using the implemented sample method.
FixedVariableComposite.sample_qubo(Q, . . .)	Sample from a QUBO using the implemented sample method.

dimod.reference.composites.fixedvariable.FixedVariableComposite.sample

FixedVariableComposite.**sample**(*bqm*, *fixed_variables=None*, ***parameters*)
Sample from the provided binary quadratic model.

Parameters

- **bqm** (*dimod.BinaryQuadraticModel*) – Binary quadratic model to be sampled from.
- **fixed_variables** (*dict*) – A dictionary of variable assignments.
- ****parameters** – Parameters for the sampling method, specified by the child sampler.

Returns *dimod.SampleSet*

dimod.reference.composites.fixedvariable.FixedVariableComposite.sample_ising

FixedVariableComposite.**sample_ising**(*h*, *J*, ***parameters*)
Sample from an Ising model using the implemented sample method.

This method is inherited from the *Sampler* base class.

Converts the Ising model into a *BinaryQuadraticModel* and then calls *sample()*.

Parameters

- **h** (*dict/list*) – Linear biases of the Ising problem. If a dict, should be of the form *{v: bias, . . . }* where is a spin-valued variable and *bias* is its associated bias. If a list, it is treated as a list of biases where the indices are the variable labels.
- **J** (*dict[(variable, variable), bias]*) – Quadratic biases of the Ising problem.
- ****kwargs** – See the implemented sampling for additional keyword definitions.

Returns *SampleSet*

See also:

sample(), *sample_qubo()*

dimod.reference.composites.fixedvariable.FixedVariableComposite.sample_qubo

FixedVariableComposite.**sample_qubo**(*Q*, ***parameters*)

Sample from a QUBO using the implemented sample method.

This method is inherited from the *Sampler* base class.

Converts the QUBO into a *BinaryQuadraticModel* and then calls *sample()*.

Parameters

- **Q** (*dict*) – Coefficients of a quadratic unconstrained binary optimization (QUBO) problem. Should be a dict of the form *{(u, v): bias, . . . }* where *u*, *v*, are binary-valued variables and *bias* is their associated coefficient.
- ****kwargs** – See the implemented sampling for additional keyword definitions.

Returns *SampleSet*

See also:

sample(), *sample_ising()*

Roof Duality Composite

A composite that uses the roof duality algorithm[1,2] to fix some variables in the binary quadratic model before passing it on to its child sampler.

Class

class RoofDualityComposite(*child_sampler*)

Uses roof duality to assign some variables before invoking child sampler.

Uses the *fix_variables()* function to determine variable assignments, then fixes them before calling the child sampler. Returned samples include the fixed variables.

Parameters **child** (*dimod.Sampler*) – A dimod sampler. Used to sample the bqm after variables have been fixed.

[1] Boros, E., P.L. Hammer, G. Tavares. Preprocessing of Unconstrained Quadratic Binary Optimization. Rutcor Research Report 10-2006, April, 2006.

[2] Boros, E., P.L. Hammer. Pseudo-Boolean optimization. Discrete Applied Mathematics 123, (2002), pp. 155-225

Properties

RoofDualityComposite.child	The child sampler.
RoofDualityComposite.children	List of child samplers that that are used by this composite.
RoofDualityComposite.parameters	A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevent to each parameter.
RoofDualityComposite.properties	A dict containing any additional information about the sampler.

dimod.reference.composites.roofduality.RoofDualityComposite.child

RoofDualityComposite.**child**

The child sampler. First sampler in `Composite.children`.

Type *Sampler*

dimod.reference.composites.roofduality.RoofDualityComposite.children

RoofDualityComposite.**children**

List of child samplers that that are used by this composite.

Type list[*Sampler*]

dimod.reference.composites.roofduality.RoofDualityComposite.parameters

RoofDualityComposite.**parameters**

A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevent to each parameter.

Type dict

dimod.reference.composites.roofduality.RoofDualityComposite.properties

RoofDualityComposite.**properties**

A dict containing any additional information about the sampler.

Type dict

Methods

RoofDualityComposite.sample(bqm[, sampling_mode])	Sample from the provided binary quadratic model.
RoofDualityComposite.sample_ising(h, J, ...)	Sample from an Ising model using the implemented sample method.
RoofDualityComposite.sample_qubo(Q, **parameters)	Sample from a QUBO using the implemented sample method.

dimod.reference.composites.roofduality.RoofDualityComposite.sample

RoofDualityComposite.**sample**(*bqm*, *sampling_mode=True*, ***parameters*)

Sample from the provided binary quadratic model.

Uses the `fix_variables()` function to determine which variables to fix.

Parameters

- **bqm** (`dimod.BinaryQuadraticModel`) – Binary quadratic model to be sampled from.
- **sampling_mode** (`bool, optional, default=True`) – In sampling mode, only roof-duality is used. When *sampling_mode* is false, strongly connected components are used to fix more variables, but in some optimal solutions these variables may take different values.
- ****parameters** – Parameters for the child sampler.

Returns `dimod.SampleSet`

dimod.reference.composites.roofduality.RoofDualityComposite.sample_ising

RoofDualityComposite.**sample_ising**(*h*, *J*, ***parameters*)

Sample from an Ising model using the implemented sample method.

This method is inherited from the `Sampler` base class.

Converts the Ising model into a `BinaryQuadraticModel` and then calls `sample()`.

Parameters

- **h** (`dict/list`) – Linear biases of the Ising problem. If a dict, should be of the form *{v: bias, . . . }* where is a spin-valued variable and *bias* is its associated bias. If a list, it is treated as a list of biases where the indices are the variable labels.
- **J** (`dict[(variable, variable), bias]`) – Quadratic biases of the Ising problem.
- ****kwargs** – See the implemented sampling for additional keyword definitions.

Returns `SampleSet`

See also:

`sample()`, `sample_qubo()`

dimod.reference.composites.roofduality.RoofDualityComposite.sample_qubo

RoofDualityComposite.**sample_qubo**(*Q*, ***parameters*)

Sample from a QUBO using the implemented sample method.

This method is inherited from the `Sampler` base class.

Converts the QUBO into a `BinaryQuadraticModel` and then calls `sample()`.

Parameters

- **Q** (`dict`) – Coefficients of a quadratic unconstrained binary optimization (QUBO) problem. Should be a dict of the form *{(u, v): bias, . . . }* where *u*, *v*, are binary-valued variables and *bias* is their associated coefficient.
- ****kwargs** – See the implemented sampling for additional keyword definitions.

Returns *SampleSet*

See also:

sample(), *sample_ising()*

Scale Composite

A composite that scales problem variables as directed. if scalar is not given calculates it based on quadratic and bias ranges.

Class

class ScaleComposite(*child_sampler*)

Composite to scale variables of a problem

Scales the variables of a bqm and modifies linear and quadratic terms accordingly.

Parameters **sampler** (*dimod.Sampler*) – A dimod sampler

Examples

This example uses *ScaleComposite* to instantiate a composed sampler that submits a simple Ising problem to a sampler. The composed sampler scales linear, quadratic biases and offset as indicated by options.

```
>>> h = {'a': -4.0, 'b': -4.0}
>>> J = {('a', 'b'): 3.2}
>>> sampler = dimod.ScaleComposite(dimod.ExactSolver())
>>> response = sampler.sample_ising(h, J, scalar=0.5,
...                 ignored_interactions=[('a','b')])
```

Properties

ScaleComposite.child	The child sampler.
ScaleComposite.children	List of child samplers that that are used by this composite.
ScaleComposite.parameters	A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevent to each parameter.
ScaleComposite.properties	A dict containing any additional information about the sampler.

dimod.reference.composites.scalecomposite.ScaleComposite.child

ScaleComposite.**child**

The child sampler. First sampler in `Composite.children`.

Type *Sampler*

dimod.reference.composites.scalecomposite.ScaleComposite.children

ScaleComposite.**children**

List of child samplers that that are used by this composite.

Type list[*Sampler*]

dimod.reference.composites.scalecomposite.ScaleComposite.parameters

ScaleComposite.**parameters**

A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevent to each parameter.

Type dict

dimod.reference.composites.scalecomposite.ScaleComposite.properties

ScaleComposite.**properties**

A dict containing any additional information about the sampler.

Type dict

Methods

ScaleComposite.sample(bqm[, scalar, . . .])	Scale and sample from the provided binary quadratic model.
ScaleComposite.sample_ising(h, J[, offset, . . .])	Scale and sample from the problem provided by h, J, offset
ScaleComposite.sample_qubo(Q, **parameters)	Sample from a QUBO using the implemented sample method.

dimod.reference.composites.scalecomposite.ScaleComposite.sample

ScaleComposite.**sample**(*bqm*, *scalar=None*, *bias_range=1*, *quadratic_range=None*, *ignored_variables=None*, *ignored_interactions=None*, *ignore_offset=False*, ***parameters*)

Scale and sample from the provided binary quadratic model.

if scalar is not given, problem is scaled based on bias and quadratic ranges. See *BinaryQuadraticModel.scale()* and *BinaryQuadraticModel.normalize()*

Parameters

- **bqm** (*dimod.BinaryQuadraticModel*) – Binary quadratic model to be sampled from.
- **scalar** (*number*) – Value by which to scale the energy range of the binary quadratic model.
- **bias_range** (*number/pair*) – Value/range by which to normalize the all the biases, or if *quadratic_range* is provided, just the linear biases.
- **quadratic_range** (*number/pair*) – Value/range by which to normalize the quadratic biases.

- **ignored_variables** (*iterable, optional*) – Biases associated with these variables are not scaled.
- **ignored_interactions** (*iterable[tuple], optional*) – As an iterable of 2-tuples. Biases associated with these interactions are not scaled.
- **ignore_offset** (*bool, default=False*) – If True, the offset is not scaled.
- ****parameters** – Parameters for the sampling method, specified by the child sampler.

Returns *dimod.SampleSet*

dimod.reference.composites.scalecomposite.ScaleComposite.sample_ising

ScaleComposite.**sample_ising**(*h, J, offset=0, scalar=None, bias_range=1, quadratic_range=None, ignored_variables=None, ignored_interactions=None, ignore_offset=False, **parameters*)

Scale and sample from the problem provided by h, J, offset

if scalar is not given, problem is scaled based on bias and quadratic ranges.

Parameters

- **h** (*dict*) – linear biases
- **J** (*dict*) – quadratic or higher order biases
- **offset** (*float, optional*) – constant energy offset
- **scalar** (*number*) – Value by which to scale the energy range of the binary quadratic model.
- **bias_range** (*number/pair*) – Value/range by which to normalize the all the biases, or if *quadratic_range* is provided, just the linear biases.
- **quadratic_range** (*number/pair*) – Value/range by which to normalize the quadratic biases.
- **ignored_variables** (*iterable, optional*) – Biases associated with these variables are not scaled.
- **ignored_interactions** (*iterable[tuple], optional*) – As an iterable of 2-tuples. Biases associated with these interactions are not scaled.
- **ignore_offset** (*bool, default=False*) – If True, the offset is not scaled.
- ****parameters** – Parameters for the sampling method, specified by the child sampler.

Returns *dimod.SampleSet*

dimod.reference.composites.scalecomposite.ScaleComposite.sample_qubo

ScaleComposite.**sample_qubo**(*Q, **parameters*)

Sample from a QUBO using the implemented sample method.

This method is inherited from the *Sampler* base class.

Converts the QUBO into a *BinaryQuadraticModel* and then calls *sample()*.

Parameters

- **Q** (*dict*) – Coefficients of a quadratic unconstrained binary optimization (QUBO) problem. Should be a dict of the form *{(u, v): bias, . . . }* where *u*, *v*, are binary-valued variables and *bias* is their associated coefficient.
- ****kwargs** – See the implemented sampling for additional keyword definitions.

Returns *SampleSet*

See also:

sample(), *sample_ising()*

Spin Reversal Transform Composite

On the D-Wave system, coupling $J_{i,j}$ adds a small bias to qubits i and j due to leakage. This can become significant for chained qubits. Additionally, qubits are biased to some small degree in one direction or another. Applying a spin-reversal transform can improve results by reducing the impact of possible analog and systematic errors. A spin-reversal transform does not alter the Ising problem; the transform simply amounts to reinterpreting spin up as spin down, and visa-versa, for a particular spin.

Class

class SpinReversalTransformComposite(*child*)

Composite for applying spin reversal transform preprocessing.

Spin reversal transforms (or "gauge transformations") are applied by flipping the spin of variables in the Ising problem. After sampling the transformed Ising problem, the same bits are flipped in the resulting sample[3].

Parameters **sampler** – A *dimod* sampler object.

Examples

This example composes a dimod ExactSolver sampler with spin transforms then uses it to sample an Ising problem.

```
>>> # Compose the sampler
>>> base_sampler = dimod.ExactSolver()
>>> composed_sampler = dimod.SpinReversalTransformComposite(base_sampler)
>>> base_sampler in composed_sampler.children
True
>>> # Sample an Ising problem
>>> response = composed_sampler.sample_ising({'a': -0.5, 'b': 1.0}, {('a', 'b'): -
↪1})
>>> print(next(response.data()))           # doctest: +SKIP
Sample(sample={'a': 1, 'b': 1}, energy=-1.5)
```

References

Properties

[3] Andrew D. King and Catherine C. McGeoch. Algorithm engineering for a quantum annealing platform. https://arxiv.org/abs/1410.2628, 2014.

SpinReversalTransformComposite.child	The child sampler.
SpinReversalTransformComposite.children	
SpinReversalTransformComposite.parameters	
SpinReversalTransformComposite.properties	

dimod.reference.composites.spin_transform.SpinReversalTransformComposite.child

SpinReversalTransformComposite.**child**

The child sampler. First sampler in Composite.children.

Type *Sampler*

dimod.reference.composites.spin_transform.SpinReversalTransformComposite.children

SpinReversalTransformComposite.**children = None**

dimod.reference.composites.spin_transform.SpinReversalTransformComposite.parameters

SpinReversalTransformComposite.**parameters = None**

dimod.reference.composites.spin_transform.SpinReversalTransformComposite.properties

SpinReversalTransformComposite.**properties = None**

Methods

SpinReversalTransformComposite.sample(bqm[, ...])	Sample from the binary quadratic model.
SpinReversalTransformComposite.sample_ising(h, ...)	Sample from an Ising model using the implemented sample method.
SpinReversalTransformComposite.sample_qubo(Q, ...)	Sample from a QUBO using the implemented sample method.

dimod.reference.composites.spin_transform.SpinReversalTransformComposite.sample

SpinReversalTransformComposite.**sample**(*bqm*, *num_spin_reversal_transforms=2*, *spin_reversal_variables=None*, ***kwargs*)

Sample from the binary quadratic model.

Parameters

- **bqm** (*BinaryQuadraticModel*) – Binary quadratic model to be sampled from.
- **num_spin_reversal_transforms** (*integer, optional, default=2*) – Number of spin reversal transform runs.

- **spin_reversal_variables** (*list/dict, optional*) – Deprecated and no longer functional.

Returns *SampleSet*

Examples

This example runs 100 spin reversals applied to one variable of a QUBO problem.

```
>>> import dimod
...
>>> base_sampler = dimod.ExactSolver()
>>> composed_sampler = dimod.SpinReversalTransformComposite(base_sampler)
>>> Q = {('a', 'a'): -1, ('b', 'b'): -1, ('a', 'b'): 2}
>>> response = composed_sampler.sample_qubo(Q,
...               num_spin_reversal_transforms=100,
...               spin_reversal_variables={'a'})
>>> len(response)
400
>>> print(next(response.data()))           # doctest: +SKIP
Sample(sample={'a': 0, 'b': 1}, energy=-1.0)
```

dimod.reference.composites.spin_transform.SpinReversalTransformComposite.sample_ising

SpinReversalTransformComposite.**sample_ising**(*h*, *J*, ***parameters*)

Sample from an Ising model using the implemented sample method.

This method is inherited from the *Sampler* base class.

Converts the Ising model into a *BinaryQuadraticModel* and then calls *sample()*.

Parameters

- **h** (*dict/list*) – Linear biases of the Ising problem. If a dict, should be of the form *{v: bias, . . . }* where is a spin-valued variable and *bias* is its associated bias. If a list, it is treated as a list of biases where the indices are the variable labels.
- **J** (*dict[(variable, variable), bias]*) – Quadratic biases of the Ising problem.
- ****kwargs** – See the implemented sampling for additional keyword definitions.

Returns *SampleSet*

See also:

sample(), *sample_qubo()*

dimod.reference.composites.spin_transform.SpinReversalTransformComposite.sample_qubo

SpinReversalTransformComposite.**sample_qubo**(*Q*, ***parameters*)

Sample from a QUBO using the implemented sample method.

This method is inherited from the *Sampler* base class.

Converts the QUBO into a *BinaryQuadraticModel* and then calls *sample()*.

Parameters

- **Q** (*dict*) – Coefficients of a quadratic unconstrained binary optimization (QUBO) problem. Should be a dict of the form *{(u, v): bias, . . . }* where *u*, *v*, are binary-valued variables and *bias* is their associated coefficient.
- ****kwargs** – See the implemented sampling for additional keyword definitions.

Returns *SampleSet*

See also:

sample(), *sample_ising()*

Structured Composite

A composite that structures a sampler.

Class

class StructureComposite(*sampler*, *nodelist*, *edgelist*)

Creates a structured composed sampler from an unstructured sampler.

Parameters

- **Sampler** (*Sampler*) – Unstructured sampler.
- **nodelist** (*list*) – Nodes/variables allowed by the sampler formatted as a list.
- **edgelist** (*list[(node, node)]*) – Edges/interactions allowed by the sampler, formatted as a list where each edge/interaction is a 2-tuple.

Examples

This example creates a composed sampler from the unstructure dimod ExactSolver sampler. The target structure is a square graph.

```
>>> import dimod
...
>>> base_sampler = dimod.ExactSolver()
>>> node_list = [0, 1, 2, 3]
>>> edge_list = [(0, 1), (1, 2), (2, 3), (0, 3)]
>>> structured_sampler = dimod.StructureComposite(base_sampler, node_list, edge_
↪list)
>>> linear = {0: 0.0, 1: 0.0, 2: 0.0, 3: 0.0}
>>> quadratic = {(0, 1): 1.0, (1, 2): 1.0, (0, 3): 1.0, (2, 3): -1.0}
>>> bqm = dimod.BinaryQuadraticModel(linear, quadratic, 1.0, dimod.Vartype.SPIN)
>>> response = structured_sampler.sample(bqm)
>>> print(next(response.data()))
Sample(sample={0: 1, 1: -1, 2: -1, 3: -1}, energy=-1.0, num_occurrences=1)
>>> # Try giving the composed sampler a non-square model
>>> del quadratic[(0, 1)]
>>> quadratic[(0, 2)] = 1.0
>>> bqm = dimod.BinaryQuadraticModel(linear, quadratic, 1.0, dimod.Vartype.SPIN)
>>> try: response = structured_sampler.sample(bqm)    # doctest: +SKIP
... except dimod.BinaryQuadraticModelStructureError as details:
...     print(details)
...
given bqm does not match the sampler's structure
```

Properties

StructureComposite.child	The child sampler.
StructureComposite.children	
StructureComposite.parameters	A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevent to each parameter.
StructureComposite.properties	A dict containing any additional information about the sampler.

dimod.reference.composites.structure.StructureComposite.child

StructureComposite.**child**

The child sampler. First sampler in `Composite.children`.

Type *Sampler*

dimod.reference.composites.structure.StructureComposite.children

StructureComposite.**children = None**

dimod.reference.composites.structure.StructureComposite.parameters

StructureComposite.**parameters**

A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevent to each parameter.

Type dict

dimod.reference.composites.structure.StructureComposite.properties

StructureComposite.**properties**

A dict containing any additional information about the sampler.

Type dict

Methods

StructureComposite.sample(bqm, **sample_kwargs)	Sample from the binary quadratic model.
StructureComposite.sample_ising(h, J, ...)	Sample from an Ising model using the implemented sample method.
StructureComposite.sample_qubo(Q, **parameters)	Sample from a QUBO using the implemented sample method.

dimod.reference.composites.structure.StructureComposite.sample

StructureComposite.**sample**(*bqm*, ***sample_kwargs*)

Sample from the binary quadratic model.

Parameters **bqm** (*BinaryQuadraticModel*) – Binary quadratic model to be sampled from.

Returns *SampleSet*

Examples

This example submits an Ising problem to a composed sampler that uses the dimod ExactSolver only on problems structured for a K2 fully connected graph.

```
>>> import dimod
...
>>> response = dimod.StructureComposite(dimod.ExactSolver(),
...                  [0, 1], [(0, 1)]).sample_ising({0: 1, 1: 1}, {})
>>> print(next(response.data()))
Sample(sample={0: -1, 1: -1}, energy=-2.0, num_occurrences=1)
```

dimod.reference.composites.structure.StructureComposite.sample_ising

StructureComposite.**sample_ising**(*h*, *J*, ***parameters*)

Sample from an Ising model using the implemented sample method.

This method is inherited from the *Sampler* base class.

Converts the Ising model into a *BinaryQuadraticModel* and then calls *sample()*.

Parameters

- **h** (*dict/list*) – Linear biases of the Ising problem. If a dict, should be of the form *{v: bias, …}* where is a spin-valued variable and *bias* is its associated bias. If a list, it is treated as a list of biases where the indices are the variable labels.
- **J** (*dict[(variable, variable), bias]*) – Quadratic biases of the Ising problem.
- ****kwargs** – See the implemented sampling for additional keyword definitions.

Returns *SampleSet*

See also:

sample(), *sample_qubo()*

dimod.reference.composites.structure.StructureComposite.sample_qubo

StructureComposite.**sample_qubo**(*Q*, ***parameters*)

Sample from a QUBO using the implemented sample method.

This method is inherited from the *Sampler* base class.

Converts the QUBO into a *BinaryQuadraticModel* and then calls *sample()*.

Parameters

- **Q** (*dict*) – Coefficients of a quadratic unconstrained binary optimization (QUBO) problem. Should be a dict of the form *{(u, v): bias, . . . }* where *u*, *v*, are binary-valued variables and *bias* is their associated coefficient.
- ****kwargs** – See the implemented sampling for additional keyword definitions.

Returns *SampleSet*

See also:

sample(), *sample_ising()*

Tracking Composite

A composite that tracks inputs and outputs.

Class

class TrackingComposite(*child*, *copy=False*)
Composite that tracks inputs and outputs for debugging and testing.

Parameters

- **child** (*dimod.Sampler*) – A dimod sampler.
- **copy** (*bool, optional, default=False*) – If True, the inputs/outputs are copied (with `copy.deepcopy()`) before they are stored. This is useful if the child sampler mutates the values.

Examples

```
>>> sampler = dimod.TrackingComposite(dimod.RandomSampler())
>>> sampleset = sampler.sample_ising({'a': -1}, {('a', 'b'): 1},
...                                  num_reads=5)
>>> sampler.input
OrderedDict([('h', {'a': -1}), ('J', {('a', 'b'): 1}), ('num_reads', 5)])
>>> sampleset == sampler.output
True
```

If we make additional calls to the sampler, the most recent input/output are stored in *input* and *output* respectively. However, all are tracked in *inputs* and *outputs*.

```
>>> sampleset = sampler.sample_qubo({('a', 'b'): 1})
>>> sampler.input
OrderedDict([('Q', {('a', 'b'): 1})])
>>> sampler.inputs # doctest: +SKIP
[OrderedDict([('h', {'a': -1}), ('J', {('a', 'b'): 1}), ('num_reads', 5)]),
 OrderedDict([('Q', {('a', 'b'): 1})])]
```

In the case that you want to nest the tracking composite, there are two patterns for retrieving the data

```
>>> from dimod import ScaleComposite, TrackingComposite, ExactSolver
...
>>> sampler = ScaleComposite(TrackingComposite(ExactSolver()))
>>> sampler.child.inputs  # empty because we haven't called sample
[]
```

```
>>> intermediate_sampler = TrackingComposite(ExactSolver())
>>> sampler = ScaleComposite(intermediate_sampler)
>>> intermediate_sampler.inputs
[]
```

Properties

TrackingComposite.input	The most recent input to any sampling method.
TrackingComposite.inputs	All of the inputs to any sampling methods.
TrackingComposite.output	The most recent output of any sampling method.
TrackingComposite.outputs	All of the outputs from any sampling methods.
TrackingComposite.parameters	A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevent to each parameter.
TrackingComposite.properties	A dict containing any additional information about the sampler.

dimod.reference.composites.tracking.TrackingComposite.input

`TrackingComposite.`**`input`**
The most recent input to any sampling method.

dimod.reference.composites.tracking.TrackingComposite.inputs

`TrackingComposite.`**`inputs`**
All of the inputs to any sampling methods.

dimod.reference.composites.tracking.TrackingComposite.output

`TrackingComposite.`**`output`**
The most recent output of any sampling method.

dimod.reference.composites.tracking.TrackingComposite.outputs

`TrackingComposite.`**`outputs`**
All of the outputs from any sampling methods.

dimod.reference.composites.tracking.TrackingComposite.parameters

`TrackingComposite.`**`parameters`**
A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevent to each parameter.

Type dict

dimod.reference.composites.tracking.TrackingComposite.properties

TrackingComposite.**properties**

A dict containing any additional information about the sampler.

Type dict

Methods

TrackingComposite.clear()	Clear all the inputs/outputs.
TrackingComposite.sample(bqm, **parameters)	Sample from the child sampler and store the given inputs/outputs.
TrackingComposite.sample_ising(h, J, ...)	Sample from the child sampler and store the given inputs/outputs.
TrackingComposite.sample_qubo(Q, **parameters)	Sample from the child sampler and store the given inputs/outputs.

dimod.reference.composites.tracking.TrackingComposite.clear

TrackingComposite.**clear**()

Clear all the inputs/outputs.

dimod.reference.composites.tracking.TrackingComposite.sample

TrackingComposite.**sample**(*bqm*, ***parameters*)

Sample from the child sampler and store the given inputs/outputs.

The binary quadratic model and any parameters are stored in *inputs*. The returned sample set is stored in *outputs*.

Parameters

- **bqm** (*dimod.BinaryQuadraticModel*) – Binary quadratic model to be sampled from.
- ****kwargs** – Parameters for the sampling method, specified by the child sampler.

Returns *dimod.SampleSet*

dimod.reference.composites.tracking.TrackingComposite.sample_ising

TrackingComposite.**sample_ising**(*h*, *J*, ***parameters*)

Sample from the child sampler and store the given inputs/outputs.

The binary quadratic model and any parameters are stored in *inputs*. The returned sample set is stored in *outputs*.

Parameters

- **h** (*dict/list*) – Linear biases of the Ising problem. If a dict, should be of the form *{v: bias, ...}* where is a spin-valued variable and *bias* is its associated bias. If a list, it is treated as a list of biases where the indices are the variable labels.
- **J** (*dict[(variable, variable), bias]*) – Quadratic biases of the Ising problem.

- ****kwargs** – Parameters for the sampling method, specified by the child sampler.

Returns `dimod.SampleSet`

dimod.reference.composites.tracking.TrackingComposite.sample_qubo

`TrackingComposite.`**`sample_qubo`**(*Q*, ***parameters*)

Sample from the child sampler and store the given inputs/outputs.

The binary quadratic model and any parameters are stored in `inputs`. The returned sample set is stored in `outputs`.

Parameters

- **Q** (`dict`) – Coefficients of a quadratic unconstrained binary optimization (QUBO) problem. Should be a dict of the form *{(u, v): bias, …}* where *u*, *v*, are binary-valued variables and *bias* is their associated coefficient.
- ****kwargs** – Parameters for the sampling method, specified by the child sampler.

Returns `dimod.SampleSet`

Truncate Composite

A composite that truncates the response based on options provided by the user.

Class

class TrackingComposite

class TruncateComposite(*child_sampler*, *n*, *sorted_by='energy'*, *aggregate=False*)

Composite to truncate the returned samples

Inherits from `dimod.ComposedSampler`.

Post-processing is expensive and sometimes one might want to only treat the lowest energy samples. This composite layer allows one to pre-select the samples within a multi-composite pipeline

Parameters

- **child_sampler** (`dimod.Sampler`) – A dimod sampler
- **n** (`int`) – Maximum number of rows in the returned sample set.
- **sorted_by** (`str/None, optional, default='energy'`) – Selects the record field used to sort the samples before truncating. Note that sample order is maintained in the underlying array.
- **aggregate** (`bool, optional, default=False`) – If True, aggregate the samples before truncating.

Note: If aggregate is True `SampleSet.record.num_occurrences` are accumulated but no other fields are.

Properties

TruncateComposite.child	The child sampler.
TruncateComposite.children	List of child samplers that that are used by this composite.
TruncateComposite.parameters	A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevent to each parameter.
TruncateComposite.properties	A dict containing any additional information about the sampler.

dimod.reference.composites.truncatecomposite.TruncateComposite.child

TruncateComposite.**child**

The child sampler. First sampler in `Composite.children`.

Type *Sampler*

dimod.reference.composites.truncatecomposite.TruncateComposite.children

TruncateComposite.**children**

List of child samplers that that are used by this composite.

Type list[*Sampler*]

dimod.reference.composites.truncatecomposite.TruncateComposite.parameters

TruncateComposite.**parameters**

A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevent to each parameter.

Type dict

dimod.reference.composites.truncatecomposite.TruncateComposite.properties

TruncateComposite.**properties**

A dict containing any additional information about the sampler.

Type dict

Methods

TruncateComposite.sample(bqm, **kwargs)	Sample from the problem provided by bqm and truncate output.
TruncateComposite.sample_ising(h, J, . . .)	Sample from an Ising model using the implemented sample method.
TruncateComposite.sample_qubo(Q, **parameters)	Sample from a QUBO using the implemented sample method.

dimod.reference.composites.truncatecomposite.TruncateComposite.sample

TruncateComposite.**sample**(*bqm*, ***kwargs*)

Sample from the problem provided by bqm and truncate output.

Parameters

- **bqm** (*dimod.BinaryQuadraticModel*) – Binary quadratic model to be sampled from.
- ****kwargs** – Parameters for the sampling method, specified by the child sampler.

Returns *dimod.SampleSet*

dimod.reference.composites.truncatecomposite.TruncateComposite.sample_ising

TruncateComposite.**sample_ising**(*h*, *J*, ***parameters*)

Sample from an Ising model using the implemented sample method.

This method is inherited from the *Sampler* base class.

Converts the Ising model into a *BinaryQuadraticModel* and then calls *sample()*.

Parameters

- **h** (*dict/list*) – Linear biases of the Ising problem. If a dict, should be of the form *{v: bias, …}* where is a spin-valued variable and *bias* is its associated bias. If a list, it is treated as a list of biases where the indices are the variable labels.
- **J** (*dict[(variable, variable), bias]*) – Quadratic biases of the Ising problem.
- ****kwargs** – See the implemented sampling for additional keyword definitions.

Returns *SampleSet*

See also:

sample(), *sample_qubo()*

dimod.reference.composites.truncatecomposite.TruncateComposite.sample_qubo

TruncateComposite.**sample_qubo**(*Q*, ***parameters*)

Sample from a QUBO using the implemented sample method.

This method is inherited from the *Sampler* base class.

Converts the QUBO into a *BinaryQuadraticModel* and then calls *sample()*.

Parameters

- **Q** (*dict*) – Coefficients of a quadratic unconstrained binary optimization (QUBO) problem. Should be a dict of the form *{(u, v): bias, …}* where *u*, *v*, are binary-valued variables and *bias* is their associated coefficient.
- ****kwargs** – See the implemented sampling for additional keyword definitions.

Returns *SampleSet*

See also:

sample(), *sample_ising()*

Higher-Order Composites

The *dimod* package includes several example higher-order composed samplers.

HigherOrderComposite

class HigherOrderComposite(*child_sampler*)

Convert a binary quadratic model sampler to a binary polynomial sampler.

Energies of the returned samples do not include the penalties.

Parameters **sampler** (*dimod.Sampler*) – A dimod sampler

Example

This example uses *HigherOrderComposite* to instantiate a composed sampler that submits a simple Ising problem to a sampler. The composed sampler creates a bqm from a higher order problem.

```
>>> sampler = dimod.HigherOrderComposite(dimod.ExactSolver())
>>> h = {0: -0.5, 1: -0.3, 2: -0.8}
>>> J = {(0, 1, 2): -1.7}
>>> sampleset = sampler.sample_hising(h, J, discard_unsatisfied=True)
>>> sampleset.first # doctest: +SKIP
Sample(sample={0: 1, 1: 1, 2: 1},
       energy=-3.3,
       num_occurrences=1,
       penalty_satisfaction=True)
```

Properties

HigherOrderComposite.child	The child sampler.
HigherOrderComposite.children	A list containing the wrapped sampler.
HigherOrderComposite.parameters	A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevant to each parameter.
HigherOrderComposite.properties	A dict containing any additional information about the sampler.

dimod.reference.composites.HigherOrderComposite.child

HigherOrderComposite.**child**

The child sampler. First sampler in `Composite.children`.

Type *Sampler*

dimod.reference.composites.HigherOrderComposite.children

HigherOrderComposite.**children**

A list containing the wrapped sampler.

dimod.reference.composites.HigherOrderComposite.parameters

HigherOrderComposite.**parameters**

A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevant to each parameter.

Type dict

dimod.reference.composites.HigherOrderComposite.properties

HigherOrderComposite.**properties**

A dict containing any additional information about the sampler.

Type dict

Methods

HigherOrderComposite.sample_poly(poly[, ...])	Sample from the given binary polynomial.
HigherOrderComposite.sample_hising(h, J, ...)	Sample from a higher-order Ising model.
HigherOrderComposite.sample_hubo(H, **kwargs)	Sample from a higher-order unconstrained binary optimization problem.

dimod.reference.composites.HigherOrderComposite.sample_poly

HigherOrderComposite.**sample_poly**(*poly*, *penalty_strength=1.0*, *keep_penalty_variables=False*, *discard_unsatisfied=False*, ***parameters*)

Sample from the given binary polynomial.

Takes the given binary polynomial, introduces penalties, reduces the higher-order problem into a quadratic problem and sends it to its child sampler.

Parameters

- **poly** (*BinaryPolynomial*) – A binary polynomial.
- **penalty_strength** (*float, optional*) – Strength of the reduction constraint. Insufficient strength can result in the binary quadratic model not having the same minimization as the polynomial.
- **keep_penalty_variables** (*bool, optional*) – default is True. if False will remove the variables used for penalty from the samples
- **discard_unsatisfied** (*bool, optional*) – default is False. If True will discard samples that do not satisfy the penalty conditions.
- **initial_state** (*dict, optional*) – Only accepted when the child sampler accepts an initial state. The initial state is given in terms of the variables in the binary polynomial. The corresponding initial values are populated for use by the child sampler.
- ****parameters** – Parameters for the sampling method, specified by
- **child sampler.** (*the*) –

Returns *dimod.SampleSet*

dimod.reference.composites.HigherOrderComposite.sample_hising

HigherOrderComposite.**sample_hising**(*h*, *J*, ***kwargs*)

Sample from a higher-order Ising model.

Convert the given higher-order Ising model to a *BinaryPolynomial* and call *sample_poly()*.

Parameters

- **h** (*dict*) – Variable biases of the Ising problem as a dict of the form *{v: bias, . . . }*, where *v* is a variable in the polynomial and *bias* its associated coefficient.
- **J** (*dict*) – Interaction biases of the Ising problem as a dict of the form *{(u, v, . . .): bias}*, where *u*, *v*, are spin-valued variables in the polynomial and *bias* their associated coefficient.
- ****kwargs** – See *sample_poly()* for additional keyword definitions.

Returns *SampleSet*

See also:

sample_poly(), *sample_hubo()*

dimod.reference.composites.HigherOrderComposite.sample_hubo

HigherOrderComposite.**sample_hubo**(*H*, ***kwargs*)

Sample from a higher-order unconstrained binary optimization problem.

Convert the given higher-order unconstrained binary optimization problem to a *BinaryPolynomial* and then call *sample_poly()*.

Parameters

- **H** (*dict*) – Coefficients of the HUBO as a dict of the form {(u, v, . . .): bias, . . . }, where *u*, *v*, are binary-valued variables in the polynomial and *bias* their associated coefficient.
- ****kwargs** – See *sample_poly()* for additional keyword definitions.

Returns *SampleSet*

See also:

sample_poly(), *sample_hising()*

PolyFixedVariableComposite

class PolyFixedVariableComposite(*child_sampler*)

Composite to fix variables of a problem to provided.

Fixes variables of a binary polynomial and modifies linear and k-local terms accordingly. Returned samples include the fixed variable

Parameters sampler (*dimod.PolySampler*) – A dimod polynomial sampler.

Examples

This example uses *PolyFixedVariableComposite* to instantiate a composed sampler that submits a simple high order Ising problem to a sampler. The composed sampler fixes a variable and modifies linear and k-local terms biases according.

```
>>> h = {1: -1.3, 2: 1.2, 3: -3.4, 4: -0.5}
>>> J = {(1, 4): -0.6, (1, 2, 3): 0.2, (1, 2, 3, 4): -0.1}
>>> poly = dimod.BinaryPolynomial.from_hising(h, J, offset=0)
>>> sampler = dimod.PolyFixedVariableComposite(dimod.ExactPolySolver())
>>> sampleset = sampler.sample_poly(poly, fixed_variables={3: -1, 4: 1})
```

Properties

PolyFixedVariableComposite.child	The child sampler.
PolyFixedVariableComposite.children	List of child samplers that that are used by this composite.
PolyFixedVariableComposite.parameters	A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevant to each parameter.
PolyFixedVariableComposite.properties	A dict containing any additional information about the sampler.

dimod.reference.composites.PolyFixedVariableComposite.child

PolyFixedVariableComposite.**child**

The child sampler. First sampler in `Composite.children`.

Type *Sampler*

dimod.reference.composites.PolyFixedVariableComposite.children

PolyFixedVariableComposite.**children**

List of child samplers that that are used by this composite.

Type list[*Sampler*]

dimod.reference.composites.PolyFixedVariableComposite.parameters

PolyFixedVariableComposite.**parameters**

A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevant to each parameter.

Type dict

dimod.reference.composites.PolyFixedVariableComposite.properties

PolyFixedVariableComposite.**properties**

A dict containing any additional information about the sampler.

Type dict

Methods

`PolyFixedVariableComposite.sample_poly(poly)`	Sample from the provided binary quadratic model.
`PolyFixedVariableComposite.sample_hising(h, ...)`	Sample from a higher-order Ising model.
`PolyFixedVariableComposite.sample_hubo(H, ...)`	Sample from a higher-order unconstrained binary optimization problem.

dimod.reference.composites.PolyFixedVariableComposite.sample_poly

`PolyFixedVariableComposite.`**`sample_poly`**(*poly*, *fixed_variables=None*, ***parameters*)

Sample from the provided binary quadratic model.

Parameters

- **poly** (`dimod.BinaryPolynomial`) – Binary polynomial model to be sampled from.
- **fixed_variables** (*dict*) – A dictionary of variable assignments.
- ****parameters** – Parameters for the sampling method, specified by the child sampler.

Returns *dimod.SampleSet*

dimod.reference.composites.PolyFixedVariableComposite.sample_hising

`PolyFixedVariableComposite.`**`sample_hising`**(*h*, *J*, ***kwargs*)

Sample from a higher-order Ising model.

Convert the given higher-order Ising model to a `BinaryPolynomial` and call `sample_poly()`.

Parameters

- **h** (*dict*) – Variable biases of the Ising problem as a dict of the form *{v: bias, ...}*, where *v* is a variable in the polynomial and *bias* its associated coefficient.
- **J** (*dict*) – Interaction biases of the Ising problem as a dict of the form *{(u, v, ...): bias}*, where *u*, *v*, are spin-valued variables in the polynomial and *bias* their associated coefficient.
- ****kwargs** – See `sample_poly()` for additional keyword definitions.

Returns *SampleSet*

See also:

`sample_poly()`, `sample_hubo()`

dimod.reference.composites.PolyFixedVariableComposite.sample_hubo

`PolyFixedVariableComposite.`**`sample_hubo`**(*H*, ***kwargs*)

Sample from a higher-order unconstrained binary optimization problem.

Convert the given higher-order unconstrained binary optimization problem to a `BinaryPolynomial` and then call `sample_poly()`.

Parameters

- **H** (*dict*) – Coefficients of the HUBO as a dict of the form {(u, v, ...): bias, ...}, where *u*, *v*, are binary-valued variables in the polynomial and *bias* their associated coefficient.
- ****kwargs** – See `sample_poly()` for additional keyword definitions.

Returns *SampleSet*

See also:

sample_poly(), *sample_hising()*

PolyScaleComposite

class PolyScaleComposite(*child*)

Composite to scale biases of a binary polynomial.

Parameters child (*PolySampler*) – A binary polynomial sampler.

Examples

```
>>> linear = {'a': -4.0, 'b': -4.0}
>>> quadratic = {('a', 'b'): 3.2, ('a', 'b', 'c'): 1}
>>> sampler = dimod.PolyScaleComposite(dimod.HigherOrderComposite(dimod.
↪ExactSolver()))
>>> response = sampler.sample_hising(linear, quadratic, scalar=0.5,
...                 ignored_terms=[('a','b')])
```

Properties

PolyScaleComposite.child	The child sampler.
PolyScaleComposite.children	The child sampler in a list
PolyScaleComposite.parameters	A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevant to each parameter.
PolyScaleComposite.properties	A dict containing any additional information about the sampler.

dimod.reference.composites.PolyScaleComposite.child

PolyScaleComposite.**child**

The child sampler. First sampler in `Composite.children`.

Type *Sampler*

dimod.reference.composites.PolyScaleComposite.children

PolyScaleComposite.**children**

The child sampler in a list

dimod.reference.composites.PolyScaleComposite.parameters

PolyScaleComposite.**parameters**

A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevant to each parameter.

Type dict

dimod.reference.composites.PolyScaleComposite.properties

PolyScaleComposite.**properties**
A dict containing any additional information about the sampler.

Type dict

Methods

PolyScaleComposite.sample_poly(poly[, ...])	Scale and sample from the given binary polynomial.
PolyScaleComposite.sample_hising(h, J, **kwargs)	Sample from a higher-order Ising model.
PolyScaleComposite.sample_hubo(H, **kwargs)	Sample from a higher-order unconstrained binary optimization problem.

dimod.reference.composites.PolyScaleComposite.sample_poly

PolyScaleComposite.**sample_poly**(*poly*, *scalar=None*, *bias_range=1*, *poly_range=None*, *ignored_terms=None*, ***parameters*)
Scale and sample from the given binary polynomial.

If scalar is not given, problem is scaled based on bias and polynomial ranges. See `BinaryPolynomial.scale()` and `BinaryPolynomial.normalize()`

Parameters

- **(obj** (*poly*) – *.BinaryPolynomial*): A binary polynomial.
- **scalar** (*number, optional*) – Value by which to scale the energy range of the binary polynomial.
- **bias_range** (*number/pair, optional, default=1*) – Value/range by which to normalize the all the biases, or if *poly_range* is provided, just the linear biases.
- **poly_range** (*number/pair, optional*) – Value/range by which to normalize the higher order biases.
- **ignored_terms** (*iterable, optional*) – Biases associated with these terms are not scaled.
- ****parameters** – Other parameters for the sampling method, specified by the child sampler.

dimod.reference.composites.PolyScaleComposite.sample_hising

PolyScaleComposite.**sample_hising**(*h*, *J*, ***kwargs*)
Sample from a higher-order Ising model.

Convert the given higher-order Ising model to a `BinaryPolynomial` and call `sample_poly()`.

Parameters

- **h** (*dict*) – Variable biases of the Ising problem as a dict of the form *{v: bias, . . . }*, where *v* is a variable in the polynomial and *bias* its associated coefficient.
- **J** (*dict*) – Interaction biases of the Ising problem as a dict of the form *{(u, v, . . .): bias}*, where *u*, *v*, are spin-valued variables in the polynomial and *bias* their associated coefficient.
- ****kwargs** – See `sample_poly()` for additional keyword definitions.

Returns `SampleSet`

See also:

`sample_poly()`, `sample_hubo()`

dimod.reference.composites.PolyScaleComposite.sample_hubo

PolyScaleComposite.**sample_hubo**(*H*, ***kwargs*)

Sample from a higher-order unconstrained binary optimization problem.

Convert the given higher-order unconstrained binary optimization problem to a `BinaryPolynomial` and then call `sample_poly()`.

Parameters

- **H** (*dict*) – Coefficients of the HUBO as a dict of the form {(u, v, . . .): bias, . . . }, where *u*, *v*, are binary-valued variables in the polynomial and *bias* their associated coefficient.
- ****kwargs** – See `sample_poly()` for additional keyword definitions.

Returns `SampleSet`

See also:

`sample_poly()`, `sample_hising()`

PolyTruncateComposite

class PolyTruncateComposite(*child_sampler*, *n*, *sorted_by='energy'*, *aggregate=False*)

Composite to truncate the returned samples

Post-processing is expensive and sometimes one might want to only treat the lowest energy samples. This composite layer allows one to pre-select the samples within a multi-composite pipeline

Parameters

- **child_sampler** (*dimod.PolySampler*) – A dimod binary polynomial sampler.
- **n** (*int*) – Maximum number of rows in the returned sample set.
- **sorted_by** (*str/None, optional, default='energy'*) – Selects the record field used to sort the samples before truncating. Note that sample order is maintained in the underlying array.
- **aggregate** (*bool, optional, default=False*) – If True, aggregate the samples before truncating.

Note: If aggregate is True `SampleSet.record.num_occurrences` are accumulated but no other fields are.

Properties

PolyTruncateComposite.child	The child sampler.
PolyTruncateComposite.children	List of child samplers that that are used by this composite.
PolyTruncateComposite.parameters	A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevant to each parameter.
PolyTruncateComposite.properties	A dict containing any additional information about the sampler.

dimod.reference.composites.PolyTruncateComposite.child

PolyTruncateComposite.**child**

The child sampler. First sampler in `Composite.children`.

Type *Sampler*

dimod.reference.composites.PolyTruncateComposite.children

PolyTruncateComposite.**children**

List of child samplers that that are used by this composite.

Type list[*Sampler*]

dimod.reference.composites.PolyTruncateComposite.parameters

PolyTruncateComposite.**parameters**

A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevant to each parameter.

Type dict

dimod.reference.composites.PolyTruncateComposite.properties

PolyTruncateComposite.**properties**

A dict containing any additional information about the sampler.

Type dict

Methods

PolyTruncateComposite.sample_poly(poly, **kwargs)	Sample from the binary polynomial and truncate output.
PolyTruncateComposite.sample_hising(h, J, ...)	Sample from a higher-order Ising model.
PolyTruncateComposite.sample_hubo(H, **kwargs)	Sample from a higher-order unconstrained binary optimization problem.

dimod.reference.composites.PolyTruncateComposite.sample_poly

`PolyTruncateComposite.sample_poly(poly, **kwargs)`

Sample from the binary polynomial and truncate output.

Parameters

- **(obj** (*poly*) – *.BinaryPolynomial*): A binary polynomial.
- ****kwargs** – Parameters for the sampling method, specified by the child sampler.

Returns *dimod.SampleSet*

dimod.reference.composites.PolyTruncateComposite.sample_hising

`PolyTruncateComposite.sample_hising(h, J, **kwargs)`

Sample from a higher-order Ising model.

Convert the given higher-order Ising model to a *BinaryPolynomial* and call *sample_poly()*.

Parameters

- **h** (*dict*) – Variable biases of the Ising problem as a dict of the form *{v: bias, …}*, where *v* is a variable in the polynomial and *bias* its associated coefficient.
- **J** (*dict*) – Interaction biases of the Ising problem as a dict of the form *{(u, v, …): bias}*, where *u*, *v*, are spin-valued variables in the polynomial and *bias* their associated coefficient.
- ****kwargs** – See *sample_poly()* for additional keyword definitions.

Returns *SampleSet*

See also:

sample_poly(), *sample_hubo()*

dimod.reference.composites.PolyTruncateComposite.sample_hubo

`PolyTruncateComposite.sample_hubo(H, **kwargs)`

Sample from a higher-order unconstrained binary optimization problem.

Convert the given higher-order unconstrained binary optimization problem to a *BinaryPolynomial* and then call *sample_poly()*.

Parameters

- **H** (*dict*) – Coefficients of the HUBO as a dict of the form {(u, v, …): bias, …}, where *u*, *v*, are binary-valued variables in the polynomial and *bias* their associated coefficient.
- ****kwargs** – See *sample_poly()* for additional keyword definitions.

Returns *SampleSet*

See also:

sample_poly(), *sample_hising()*

API for Samplers and Composites

You can create your own samplers with dimod's *Sampler* abstract base class (ABC) providing complementary methods (e.g., 'sample_qubo' if only 'sample_ising' is implemented), consistent responses, etc.

Properties of dimod Sampler Abstract Base Classes

The following table describes the inheritance, properties, methods/mixins of sampler ABCs.

ABC	Inherits from	Abstract Properties	Abstract Methods	Mixins
Sampler		*parameters*, *properties*	at least one of *sample()*, *sample_ising()*, *sample_qubo()*	*sample()*, *sample_ising()*, *sample_qubo()*
Structured		*nodelist*, *edgelist*		*structure*, *adjacency*
Composite		*children*		*child*
ComposedSampler	*Sampler*, *Composite*	*parameters*, *properties*, *children*	at least one of *sample()*, *sample_ising()*, *sample_qubo()*	*sample()*, *sample_ising()*, *sample_qubo()*, *child*
PolySampler		*parameters*, *properties*	*sample_poly()*	*sample_hising()*, *sample_hubo()*
ComposedPolySampler	*PolySampler*, *Composite*	*parameters*, *properties*, *children*	sample_poly()	sample_hising(), sample_hubo(), *child*

The table shows, for example, that the *Sampler* class requires that you implement the *parameters* and *properties* properties and at least one sampler method; the class provides the unimplemented methods as mixins.

Creating a Sampler

The *Sampler* abstract base class (see abc) helps you create new dimod samplers.

Any new dimod sampler must define a subclass of *Sampler* that implements abstract properties *parameters* and *properties* and one of the abstract methods *sample()*, *sample_ising()*, or *sample_qubo()*. The *Sampler* class provides the complementary methods as mixins and ensures consistent responses.

For example, the following steps show how to easily create a dimod sampler. It is sufficient to implement a single method (in this example the `sample_ising()` method) to create a dimod sampler with the *Sampler* class.

```
class LinearIsingSampler(dimod.Sampler):

    def sample_ising(self, h, J):
        sample = linear_ising(h, J)
        energy = dimod.ising_energy(sample, h, J)
        return dimod.SampleSet.from_samples([sample], energy=[energy])

    @property
    def properties(self):
        return dict()

    @property
    def parameters(self):
        return dict()
```

For this example, the implemented sampler *sample_ising()* can be based on a simple placeholder function, which returns a sample that minimizes the linear terms:

```
def linear_ising(h, J):
    sample = {}
    for v in h:
        if h[v] < 0:
            sample[v] = +1
        else:
            sample[v] = -1
    return sample
```

The *Sampler* ABC provides the other sample methods "for free" as mixins.

```
sampler = LinearIsingSampler()
response = sampler.sample_ising({'a': -1}, {})  # Implemented by class␣
↪LinearIsingSampler
response = sampler.sample_qubo({('a', 'a'): 1})  # Mixin provided by Sampler class
response = sampler.sample(BinaryQuadraticModel.from_ising({'a': -1}, {}))  # Mixin␣
↪provided by Sampler class
```

Below is a more complex version of the same sampler, where the `properties` and `parameters` properties return non-empty dicts.

```
class FancyLinearIsingSampler(dimod.Sampler):
    def __init__(self):
        self._properties = {'description': 'a simple sampler that only considers the␣
↪linear terms'}
        self._parameters = {'verbose': []}

    def sample_ising(self, h, J, verbose=False):
        sample = linear_ising(h, J)
        energy = dimod.ising_energy(sample, h, J)
        if verbose:
            print(sample)
        return dimod.SampleSet.from_samples([sample], energy=[energy])

    @property
    def properties(self):
        return self._properties

    @property
    def parameters(self):
        return self._parameters
```

class Sampler

Abstract base class for dimod samplers.

Provides all methods *sample()*, *sample_ising()*, *sample_qubo()* assuming at least one is implemented.

Abstract Properties

Sampler.parameters	A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevent to each parameter.

Continued on next page

Table 48 – continued from previous page

Sampler.properties	A dict containing any additional information about the sampler.

dimod.Sampler.parameters

Sampler.**parameters**

A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevent to each parameter.

Type dict

dimod.Sampler.properties

Sampler.**properties**

A dict containing any additional information about the sampler.

Type dict

Mixin Methods

Sampler.sample(bqm, **parameters)	Sample from a binary quadratic model.
Sampler.sample_ising(h, J, **parameters)	Sample from an Ising model using the implemented sample method.
Sampler.sample_qubo(Q, **parameters)	Sample from a QUBO using the implemented sample method.

dimod.Sampler.sample

Sampler.**sample**(*bqm*, ***parameters*)

Sample from a binary quadratic model.

This method is inherited from the *Sampler* base class.

Converts the binary quadratic model to either Ising or QUBO format and then invokes an implemented sampling method (one of *sample_ising()* or *sample_qubo()*).

:param *BinaryQuadraticModel*: A binary quadratic model. :param **kwargs: See the implemented sampling for additional keyword definitions.

Returns *SampleSet*

See also:

sample_ising(), *sample_qubo()*

dimod.Sampler.sample_ising

Sampler.**sample_ising**(*h*, *J*, ***parameters*)

Sample from an Ising model using the implemented sample method.

This method is inherited from the *Sampler* base class.

Converts the Ising model into a *BinaryQuadraticModel* and then calls *sample()*.

Parameters

- **h** (*dict/list*) – Linear biases of the Ising problem. If a dict, should be of the form *{v: bias, …}* where is a spin-valued variable and *bias* is its associated bias. If a list, it is treated as a list of biases where the indices are the variable labels.
- **J** (*dict[(variable, variable), bias]*) – Quadratic biases of the Ising problem.
- ****kwargs** – See the implemented sampling for additional keyword definitions.

Returns `SampleSet`

See also:

`sample()`, `sample_qubo()`

dimod.Sampler.sample_qubo

Sampler.**sample_qubo**(*Q*, ***parameters*)

Sample from a QUBO using the implemented sample method.

This method is inherited from the `Sampler` base class.

Converts the QUBO into a `BinaryQuadraticModel` and then calls `sample()`.

Parameters

- **Q** (*dict*) – Coefficients of a quadratic unconstrained binary optimization (QUBO) problem. Should be a dict of the form *{(u, v): bias, …}* where *u*, *v*, are binary-valued variables and *bias* is their associated coefficient.
- ****kwargs** – See the implemented sampling for additional keyword definitions.

Returns `SampleSet`

See also:

`sample()`, `sample_ising()`

Creating a Composed Sampler

Samplers can be composed. The composite pattern allows layers of pre- and post-processing to be applied to binary quadratic programs without needing to change the underlying sampler implementation.

We refer to these layers as *composites*. Each composed sampler must include at least one sampler, and possibly many composites.

Each composed sampler is itself a dimod sampler with all of the included methods and parameters. In this way complex samplers can be constructed.

The dimod `ComposedSampler` abstract base class inherits from `Sampler` class its abstract methods, properties, and mixins (for example, a *sample_Ising* method) and from `Composite` class the *children* property and *child* mixin (*children* being a list of supported samplers with *child* providing the first).

Examples

The dimod package's spin_transform.py reference example creates a composed sampler, *SpinReversalTransformComposite(Sampler, Composite)*, that performs spin reversal transforms ("gauge transformations") as a preprocessing step for a given sampler. The reference example implements the pseudocode below:

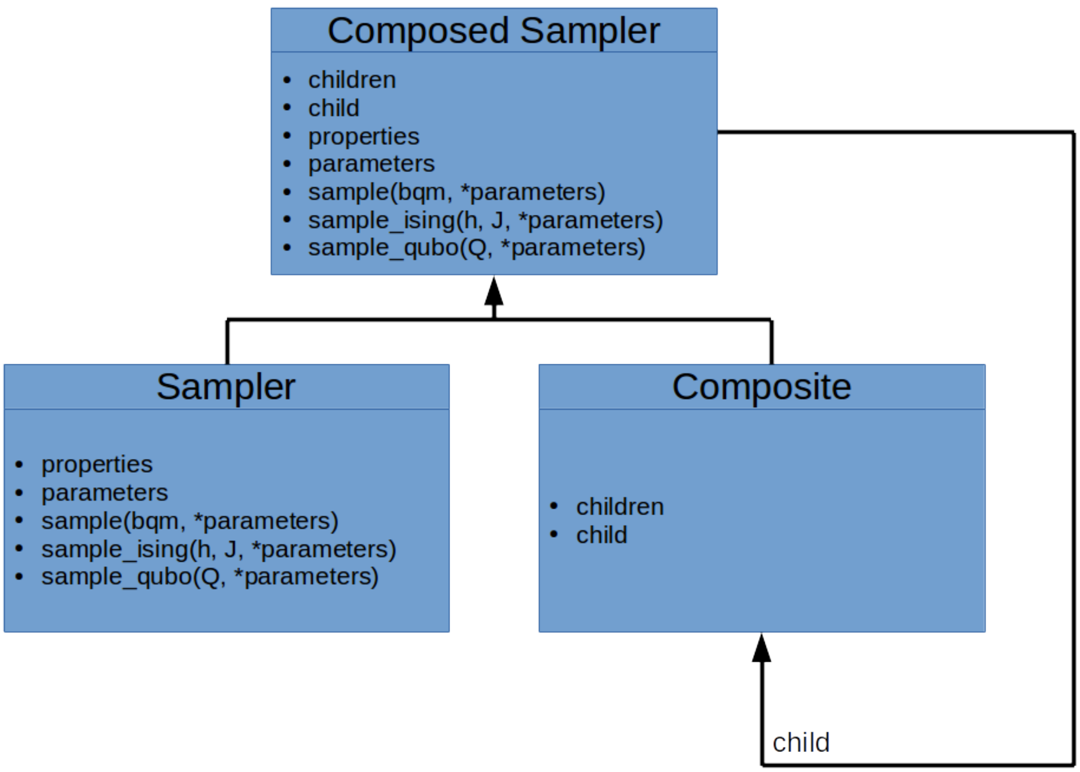

Fig. 1: Composite Pattern

```
class SpinReversalTransformComposite(Sampler, Composite):

    # Updates to inherited sampler properties and parameters
    # Definition of the composite's children (i.e., supported samplers):
    children = None
    def __init__(self, child):
        self.children = [child]

    # The composite's implementation of spin-transformation functionality:
    def sample(self, bqm, num_spin_reversal_transforms=2, spin_reversal_
↪variables=None, **kwargs):
        response = None
        # Preprocessing code that includes instantiation of a sampler:
        # flipped_response = self.child.sample(bqm, **kwargs)
        return response
```

Given a sampler, *sampler1*, the composed sampler is used as any dimod sampler. For example, the composed sampler inherits an Ising sampling method:

```
>>> composed_sampler = dimod.SpinReversalTransformComposite(sampler1) # doctest: +SKIP
>>> h = {0: -1, 1: 1} # doctest: +SKIP
>>> response = composed_sampler.sample_ising(h, {}) # doctest: +SKIP
```

class ComposedSampler

Abstract base class for dimod composed samplers.

Inherits from *Sampler* and *Composite*.

class Composite

Abstract base class for dimod composites.

Provides the *Composite.child* mixin property and defines the *Composite.children* abstract property to be implemented. These define the supported samplers for the composed sampler.

Abstract Properties

Composite.children	List of child samplers that that are used by this composite.

dimod.Composite.children

Composite.**children**

List of child samplers that that are used by this composite.

Type list[*Sampler*]

Mixin Properties

Composite.child	The child sampler.

dimod.Composite.child

`Composite.child`
The child sampler. First sampler in *Composite.children*.

Type *Sampler*

Creating a Structured Sampler

A structured sampler can only sample from binary quadratic models with a specific graph.

For structured samplers you must implement the *nodelist* and *edgelist* properties. The *Structured* abstract base class provides access to the *structure* and *adjacency* properties as well as any method or properties required by the *Sampler* abstract base class. The *bqm_structured* decorator verifies that any given binary quadratic model conforms to the supported structure.

Examples

This simple example shows a structured sampler that can only sample from a binary quadratic model with two variables and one interaction.

```
class TwoVariablesSampler(dimod.Sampler, dimod.Structured):
    @property
    def nodelist(self):
        return [0, 1]

    @property
    def edgelist(self):
        return [(0, 1)]

    @property
    def properties(self):
        return dict()

    @property
    def parameters(self):
        return dict()

    @dimod.decorators.bqm_structured
    def sample(self, bqm):
        # All bqm's passed in will be a subgraph of the sampler's structure
        variable_list = list(bqm.linear)
        samples = []
        energies = []
        for values in itertools.product(bqm.vartype.value, repeat=len(bqm)):
            sample = dict(zip(variable_list, values))
            samples.append(sample)
            energies.append(bqm.energy(sample))

        return dimod.SampleSet.from_samples(samples, bqm.Vartype, energies)

        return response
```

class Structured
The abstract base class for dimod structured samplers.

Provides the *Structured.adjacency* and *Structured.structure* properties.

Abstract properties *nodelist* and *edgelist* must be implemented.

Abstract Properties

Structured.nodelist	Nodes/variables allowed by the sampler.
Structured.edgelist	Edges/interactions allowed by the sampler in the form *[(u, v), . . .]*.

dimod.Structured.nodelist

`Structured.nodelist`

Nodes/variables allowed by the sampler.

Type list

dimod.Structured.edgelist

`Structured.edgelist`

Edges/interactions allowed by the sampler in the form *[(u, v), . . .]*.

Type list

Mixin Properties

Structured.adjacency	Adjacency structure formatted as a dict, where keys are the nodes of the structured sampler and values are sets of all adjacent nodes for each key node.
Structured.structure	Structure of the structured sampler formatted as a `namedtuple`, *Structure(nodelist, edgelist, adjacency)*, where the 3-tuple values are the *nodelist*, *edgelist* and *adjacency* attributes.

dimod.Structured.adjacency

`Structured.adjacency`

Adjacency structure formatted as a dict, where keys are the nodes of the structured sampler and values are sets of all adjacent nodes for each key node.

Type dict[variable, set]

dimod.Structured.structure

`Structured.structure`

Structure of the structured sampler formatted as a `namedtuple`, *Structure(nodelist, edgelist, adjacency)*, where the 3-tuple values are the *nodelist*, *edgelist* and *adjacency* attributes.

Creating a Binary Polynomial Sampler

Samplers that handle binary polynomials: problems with binary variables that are not constrained to quadratic interactions.

class PolySampler

Sampler that supports binary polynomials.

Binary polynomials are an extension of binary quadratic models that allow higher-order interactions.

Abstract Properties

PolySampler.parameters	A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevant to each parameter.
PolySampler.properties	A dict containing any additional information about the sampler.

dimod.PolySampler.parameters

PolySampler.**parameters**

A dict where keys are the keyword parameters accepted by the sampler methods and values are lists of the properties relevant to each parameter.

Type dict

dimod.PolySampler.properties

PolySampler.**properties**

A dict containing any additional information about the sampler.

Type dict

Abstract Methods

PolySampler.sample_poly(polynomial, **kwargs)	Sample from a higher-order polynomial.

dimod.PolySampler.sample_poly

PolySampler.**sample_poly**(*polynomial*, ***kwargs*)

Sample from a higher-order polynomial.

Mixin Methods

PolySampler.sample_hising(h, J, **kwargs)	Sample from a higher-order Ising model.

Continued on next page

Table 56 – continued from previous page

PolySampler.sample_hubo(H, **kwargs)	Sample from a higher-order unconstrained binary optimization problem.

dimod.PolySampler.sample_hising

PolySampler.**sample_hising**(*h*, *J*, ***kwargs*)
Sample from a higher-order Ising model.

Convert the given higher-order Ising model to a *BinaryPolynomial* and call *sample_poly()*.

Parameters

- **h** (*dict*) – Variable biases of the Ising problem as a dict of the form *{v: bias, …}*, where *v* is a variable in the polynomial and *bias* its associated coefficient.
- **J** (*dict*) – Interaction biases of the Ising problem as a dict of the form *{(u, v, …): bias}*, where *u*, *v*, are spin-valued variables in the polynomial and *bias* their associated coefficient.
- ****kwargs** – See *sample_poly()* for additional keyword definitions.

Returns *SampleSet*

See also:

sample_poly(), *sample_hubo()*

dimod.PolySampler.sample_hubo

PolySampler.**sample_hubo**(*H*, ***kwargs*)
Sample from a higher-order unconstrained binary optimization problem.

Convert the given higher-order unconstrained binary optimization problem to a *BinaryPolynomial* and then call *sample_poly()*.

Parameters

- **H** (*dict*) – Coefficients of the HUBO as a dict of the form {(u, v, …): bias, …}, where *u*, *v*, are binary-valued variables in the polynomial and *bias* their associated coefficient.
- ****kwargs** – See *sample_poly()* for additional keyword definitions.

Returns *SampleSet*

See also:

sample_poly(), *sample_hising()*

Creating a Composed Binary Polynomial Sampler

class ComposedPolySampler
Abstract base class for dimod composed polynomial samplers.

Inherits from *PolySampler* and *Composite*.

2.2.3 Samples

dimod *sampler*s sample from a problem's objective function, such as a BQM, and return an iterable of samples contained in a `SampleSet` class. In addition to containing the returned solutions and some additional information, and providing methods to work with the solutions, `SampleSet` is also used, for example, by dwave-hybrid, which iterates sets of samples through samplers to solve arbitrary QUBOs. dimod provides functionality for creating and manipulating samples.

sample_like Objects

`as_samples`(samples_like[, dtype, copy, order])	Convert a samples_like object to a NumPy array and list of labels.

dimod.as_samples

as_samples (*samples_like*, *dtype=None*, *copy=False*, *order='C'*)

Convert a samples_like object to a NumPy array and list of labels.

Parameters

- **samples_like** (*samples_like*) – A collection of raw samples. *samples_like* is an extension of NumPy's array_like structure. See examples below.
- **dtype** (*data-type, optional*) – dtype for the returned samples array. If not provided, it is either derived from *samples_like*, if that object has a dtype, or set to `numpy.int8`.
- **copy** (*bool, optional, default=False*) – If true, then samples_like is guaranteed to be copied, otherwise it is only copied if necessary.
- **order** (*{'K', 'A', 'C', 'F'}, optional, default='C'*) – Specify the memory layout of the array. See `numpy.array()`.

Returns

A 2-tuple containing:

`numpy.ndarray`: Samples.

list: Variable labels

Return type tuple

Examples

The following examples convert a variety of samples_like objects:

NumPy arrays

```
>>> import numpy as np
...
>>> dimod.as_samples(np.ones(5, dtype='int8'))
(array([[1, 1, 1, 1, 1]], dtype=int8), [0, 1, 2, 3, 4])
>>> dimod.as_samples(np.zeros((5, 2), dtype='int8'))
(array([[0, 0],
        [0, 0],
        [0, 0],
```

(continues on next page)

(continued from previous page)

```
       [0, 0],
       [0, 0]], dtype=int8), [0, 1])
```

Lists

```
>>> dimod.as_samples([-1, +1, -1])
(array([[-1,  1, -1]], dtype=int8), [0, 1, 2])
>>> dimod.as_samples([[-1], [+1], [-1]])
(array([[-1],
       [ 1],
       [-1]], dtype=int8), [0])
```

Dicts

```
>>> dimod.as_samples({'a': 0, 'b': 1, 'c': 0}) # doctest: +SKIP
(array([[0, 1, 0]], dtype=int8), ['a', 'b', 'c'])
>>> dimod.as_samples([{'a': -1, 'b': +1}, {'a': 1, 'b': 1}]) # doctest: +SKIP
(array([[-1,  1],
        [ 1,  1]], dtype=int8), ['a', 'b'])
```

A 2-tuple containing an array_like object and a list of labels

```
>>> dimod.as_samples(([-1, +1, -1], ['a', 'b', 'c']))
(array([[-1,  1, -1]], dtype=int8), ['a', 'b', 'c'])
>>> dimod.as_samples((np.zeros((5, 2), dtype='int8'), ['in', 'out']))
(array([[0, 0],
       [0, 0],
       [0, 0],
       [0, 0],
       [0, 0]], dtype=int8), ['in', 'out'])
```

SampleSet

class SampleSet(*record*, *variables*, *info*, *vartype*)

Samples and any other data returned by dimod samplers.

Parameters

- **record** (`numpy.recarray`) – A NumPy record array. Must have 'sample', 'energy' and 'num_occurrences' as fields. The 'sample' field should be a 2D NumPy array where each row is a sample and each column represents the value of a variable.
- **variables** (*iterable*) – An iterable of variable labels, corresponding to columns in *record.samples*.
- **info** (*dict*) – Information about the `SampleSet` as a whole, formatted as a dict.
- **vartype** (`Vartype`/str/set) – Variable type for the `SampleSet`. Accepted input values:
  - `Vartype.SPIN`, `'SPIN'`, `{-1, 1}`
  - `Vartype.BINARY`, `'BINARY'`, `{0, 1}`

Examples

This example creates a SampleSet out of a samples_like object (a NumPy array).

```
>>> import dimod
>>> import numpy as np
...
>>> dimod.SampleSet.from_samples(np.ones(5, dtype='int8'), 'BINARY', 0)    #␣
↪doctest: +SKIP
SampleSet(rec.array([([1, 1, 1, 1, 1], 0, 1)],
...         dtype=[('sample', 'i1', (5,)), ('energy', '<i4'), ('num_occurrences', '
↪<i4')]),
...         [0, 1, 2, 3, 4], {}, 'BINARY')
```

Properties

SampleSet.first	Sample with the lowest-energy.
SampleSet.info	Dict of information about the *SampleSet* as a whole.
SampleSet.record	numpy.recarray containing the samples, energies, number of occurences, and other sample data.
SampleSet.variables	VariableIndexView of variable labels.
SampleSet.vartype	*Vartype* of the samples.

dimod.SampleSet.first

SampleSet.**first**

Sample with the lowest-energy.

Raises ValueError – If empty.

Example

```
>>> sampleset = dimod.ExactSolver().sample_ising({'a': 1}, {('a', 'b'): 1})
>>> sampleset.first
Sample(sample={'a': -1, 'b': 1}, energy=-2.0, num_occurrences=1)
```

dimod.SampleSet.info

SampleSet.**info**

Dict of information about the *SampleSet* as a whole.

Examples

This example shows the type of information that might be returned by a dimod sampler by submitting a BQM that sets a value on a D-Wave system's first listed coupler.

```
>>> from dwave.system.samplers import DWaveSampler    # doctest: +SKIP
>>> sampler = DWaveSampler()    # doctest: +SKIP
>>> bqm = dimod.BQM({}, {sampler.edgelist[0]: -1}, 0, dimod.SPIN)    # doctest:␣
↪+SKIP
>>> sampler.sample(bqm).info    # doctest: +SKIP
{'timing': {'qpu_sampling_time': 315,
 'qpu_anneal_time_per_sample': 20,
```

(continues on next page)

(continued from previous page)

```
'qpu_readout_time_per_sample': 274,
# Snipped above response for brevity
```

dimod.SampleSet.record

SampleSet.**record**

numpy.recarray containing the samples, energies, number of occurences, and other sample data.

Examples

```
>>> import dimod
...
>>> sampler = dimod.ExactSolver()
>>> sampleset = sampler.sample_ising({'a': -0.5, 'b': 1.0}, {('a', 'b'): -1.0})
>>> sampleset.record
rec.array([([-1, -1], -1.5, 1), ([ 1, -1], -0.5, 1), ([ 1,  1], -0.5, 1),
           ([-1,  1],  2.5, 1)],
          dtype=[('sample', 'i1', (2,)), ('energy', '<f8'), ('num_occurrences', '
↪<i8')])
>>> sampleset.record.sample
array([[-1, -1],
       [ 1, -1],
       [ 1,  1],
       [-1,  1]], dtype=int8)
>>> sampleset.record.energy
array([-1.5, -0.5, -0.5,  2.5])
```

dimod.SampleSet.variables

SampleSet.**variables**

VariableIndexView of variable labels.

Corresponds to columns of the sample field of *SampleSet.record*.

dimod.SampleSet.vartype

SampleSet.**vartype**

Vartype of the samples.

Methods

SampleSet.aggregate()	Create a new SampleSet with repeated samples aggregated.
SampleSet.append_variables(samples_like[, ...])	Create a new sampleset with the given variables and values.
SampleSet.change_vartype(vartype[, ...])	Return the *SampleSet* with the given vartype.
SampleSet.copy()	Create a shallow copy.

Continued on next page

Table 59 – continued from previous page

SampleSet.data([fields, sorted_by, name, ...])	Iterate over the data in the *SampleSet*.
SampleSet.done()	Return True if a pending computation is done.
SampleSet.from_future(future[, result_hook])	Construct a *SampleSet* referencing the result of a future computation.
SampleSet.from_samples(samples_like, ...[, ...])	Build a *SampleSet* from raw samples.
SampleSet.from_samples_bqm(samples_like, ...)	Build a sample set from raw samples and a binary quadratic model.
SampleSet.from_serializable(obj)	Deserialize a *SampleSet*.
SampleSet.lowest([rtol, atol])	Return a sample set containing the lowest-energy samples.
SampleSet.resolve()	Ensure that the sampleset is resolved if constructed from a future.
SampleSet.relabel_variables(mapping[, inplace])	Relabel the variables of a *SampleSet* according to the specified mapping.
SampleSet.samples([n, sorted_by])	Return an iterable over the samples.
SampleSet.slice(*slice_args, **kwargs)	Create a new sample set with rows sliced according to standard Python slicing syntax.
SampleSet.to_pandas_dataframe([sample_column])	Convert a sample set to a Pandas DataFrame
SampleSet.to_serializable([use_bytes, ...])	Convert a *SampleSet* to a serializable object.
SampleSet.truncate(n[, sorted_by])	Create a new sample set with up to n rows.

dimod.SampleSet.aggregate

SampleSet.**aggregate**()

Create a new SampleSet with repeated samples aggregated.

Returns *SampleSet*

Note: `SampleSet.record.num_occurrences` are accumulated but no other fields are.

Examples

This examples aggregates a sample set with two identical samples out of three.

```
>>> sampleset = dimod.SampleSet.from_samples([[0, 0, 1], [0, 0, 1],
...                                           [1, 1, 1]],
...                                           dimod.BINARY,
...                                           [0, 0, 1])
>>> print(sampleset)
   0  1  2 energy num_oc.
0  0  0  1      0       1
1  0  0  1      0       1
2  1  1  1      1       1
['BINARY', 3 rows, 3 samples, 3 variables]
>>> print(sampleset.aggregate())
   0  1  2 energy num_oc.
0  0  0  1      0       2
1  1  1  1      1       1
['BINARY', 2 rows, 3 samples, 3 variables]
```

dimod.SampleSet.append_variables

`SampleSet.append_variables(samples_like, sort_labels=True)`
Create a new sampleset with the given variables and values.

Not defined for empty sample sets. If *sample_like* is a `SampleSet`, its data vectors and info are ignored.

Parameters

- **samples_like** – Samples to add to the sample set. Either a single sample or identical in length to the sample set. 'samples_like' is an extension of NumPy's array_like. See `as_samples()`.
- **sort_labels** (*bool, optional, default=True*) – Return `SampleSet.variables` in sorted order. For mixed (unsortable) types, the given order is maintained.

Returns New sample set with the variables/values added.

Return type `SampleSet`

Examples

```
>>> sampleset = dimod.SampleSet.from_samples([{'a': -1, 'b': +1},
...                                            {'a': +1, 'b': +1}],
...                                           dimod.SPIN,
...                                           energy=[-1.0, 1.0])
>>> new = sampleset.append_variables({'c': -1})
>>> print(new)
   a  b  c energy num_oc.
0 -1 +1 -1   -1.0       1
1 +1 +1 -1    1.0       1
['SPIN', 2 rows, 2 samples, 3 variables]
```

Add variables from another sample set to the previous example. Note that the energies remain unchanged.

```
>>> another = dimod.SampleSet.from_samples([{'c': -1, 'd': +1},
...                                          {'c': +1, 'd': +1}],
...                                         dimod.SPIN,
...                                         energy=[-2.0, 1.0])
>>> new = sampleset.append_variables(another)
>>> print(new)
   a  b  c  d energy num_oc.
0 -1 +1 -1 +1   -1.0       1
1 +1 +1 +1 +1    1.0       1
['SPIN', 2 rows, 2 samples, 4 variables]
```

dimod.SampleSet.change_vartype

`SampleSet.change_vartype(vartype, energy_offset=0.0, inplace=True)`
Return the `SampleSet` with the given vartype.

Parameters

- **vartype** (`Vartype`/str/set) – Variable type to use for the new `SampleSet`. Accepted input values:
  - `Vartype.SPIN`, `'SPIN'`, `{-1, 1}`

   – Vartype.BINARY, 'BINARY', {0, 1}

- **energy_offset** (*number, optional, defaul=0.0*) – Constant value applied to the 'energy' field of *SampleSet.record*.
- **inplace** (*bool, optional, default=True*) – If True, the instantiated *SampleSet* is updated; otherwise, a new *SampleSet* is returned.

Returns SampleSet with changed vartype. If *inplace* is True, returns itself.

Return type *SampleSet*

Examples

This example creates a binary copy of a spin-valued *SampleSet*.

```
>>> import dimod
...
>>> sampleset = dimod.ExactSolver().sample_ising({'a': -0.5, 'b': 1.0}, {('a', 'b
↪'): -1})
>>> sampleset_binary = sampleset.change_vartype(dimod.BINARY, energy_offset=1.0,␣
↪inplace=False)
>>> sampleset_binary.vartype is dimod.BINARY
True
>>> for datum in sampleset_binary.data(fields=['sample', 'energy', 'num_
↪occurrences']):    # doctest: +SKIP
...     print(datum)
Sample(sample={'a': 0, 'b': 0}, energy=-0.5, num_occurrences=1)
Sample(sample={'a': 1, 'b': 0}, energy=0.5, num_occurrences=1)
Sample(sample={'a': 1, 'b': 1}, energy=0.5, num_occurrences=1)
Sample(sample={'a': 0, 'b': 1}, energy=3.5, num_occurrences=1)
```

dimod.SampleSet.copy

SampleSet.**copy**()

Create a shallow copy.

dimod.SampleSet.data

SampleSet.**data**(*fields=None, sorted_by='energy', name='Sample', reverse=False, sample_dict_cast=True, index=False*)

Iterate over the data in the *SampleSet*.

Parameters

- **fields** (*list, optional, default=None*) – If specified, only these fields are included in the yielded tuples. The special field name 'sample' can be used to view the samples.
- **sorted_by** (*str/None, optional, default='energy'*) – Selects the record field used to sort the samples. If None, the samples are yielded in record order.
- **name** (*str/None, optional, default='Sample'*) – Name of the yielded namedtuples or None to yield regular tuples.
- **reverse** (*bool, optional, default=False*) – If True, yield in reverse order.

- **sample_dict_cast** (*bool, optional, default=True*) – Samples are returned as dicts rather than SampleView, which requires heavy memory usage. Set to False to reduce load on memory.
- **index** (*bool, optional, default=False*) – If True, *datum.idx* gives the corresponding index of the *SampleSet.record*.

Yields *namedtuple/tuple* – The data in the *SampleSet*, in the order specified by the input *fields*.

Examples

```
>>> import dimod
...
>>> sampleset = dimod.ExactSolver().sample_ising({'a': -0.5, 'b': 1.0}, {('a', 'b
↪'): -1})
>>> for datum in sampleset.data(fields=['sample', 'energy']):   # doctest: +SKIP
...     print(datum)
Sample(sample={'a': -1, 'b': -1}, energy=-1.5)
Sample(sample={'a': 1, 'b': -1}, energy=-0.5)
Sample(sample={'a': 1, 'b': 1}, energy=-0.5)
Sample(sample={'a': -1, 'b': 1}, energy=2.5)
>>> for energy, in sampleset.data(fields=['energy'], sorted_by='energy'):
...     print(energy)
...
-1.5
-0.5
-0.5
2.5
>>> print(next(sampleset.data(fields=['energy'], name='ExactSolverSample')))
ExactSolverSample(energy=-1.5)
```

dimod.SampleSet.done

SampleSet.**done**()

Return True if a pending computation is done.

Used when a *SampleSet* is constructed with *SampleSet.from_future()*.

Examples

This example uses a Future object directly. Typically a Executor sets the result of the future (see documentation for concurrent.futures).

```
>>> import dimod
>>> from concurrent.futures import Future
...
>>> future = Future()
>>> sampleset = dimod.SampleSet.from_future(future)
>>> future.done()
False
>>> future.set_result(dimod.ExactSolver().sample_ising({0: -1}, {}))
>>> future.done()
True
>>> sampleset.record.sample
```

(continues on next page)

(continued from previous page)

```
array([[-1],
       [ 1]], dtype=int8)
```

dimod.SampleSet.from_future

classmethod SampleSet.**from_future**(*future*, *result_hook=None*)

Construct a *SampleSet* referencing the result of a future computation.

Parameters

- **future** (*object*) – Object that contains or will contain the information needed to construct a *SampleSet*. If *future* has a done() method, this determines the value returned by *SampleSet.done()*.
- **result_hook** (*callable, optional*) – A function that is called to resolve the future. Must accept the future and return a *SampleSet*. If not provided, set to

```
def result_hook(future):
    return future.result()
```

Returns *SampleSet*

Notes

The future is resolved on the first read of any of the *SampleSet* properties.

Examples

Run a dimod sampler on a single thread and load the returned future into *SampleSet*.

```
>>> import dimod
>>> from concurrent.futures import ThreadPoolExecutor
...
>>> bqm = dimod.BinaryQuadraticModel.from_ising({}, {('a', 'b'): -1})
>>> with ThreadPoolExecutor(max_workers=1) as executor:
...     future = executor.submit(dimod.ExactSolver().sample, bqm)
...     sampleset = dimod.SampleSet.from_future(future)
>>> sampleset.record
rec.array([([-1, -1], -1., 1), ([ 1, -1],  1., 1), ([ 1,  1], -1., 1),
           ([-1,  1],  1., 1)],
          dtype=[('sample', 'i1', (2,)), ('energy', '<f8'), ('num_occurrences', '
↪<i8')])
```

dimod.SampleSet.from_samples

classmethod SampleSet.**from_samples**(*samples_like*, *vartype*, *energy*, *info=None*, *num_occurrences=None*, *aggregate_samples=False*, *sort_labels=True*, ***vectors*)

Build a *SampleSet* from raw samples.

Parameters

- **samples_like** – A collection of raw samples. 'samples_like' is an extension of NumPy's array_like. See `as_samples()`.
- **vartype** (`Vartype`/str/set) – Variable type for the `SampleSet`. Accepted input values:
  - `Vartype.SPIN`, `'SPIN'`, `{-1, 1}`
  - `Vartype.BINARY`, `'BINARY'`, `{0, 1}`
- **energy** (*array_like*) – Vector of energies.
- **info** (*dict, optional*) – Information about the `SampleSet` as a whole formatted as a dict.
- **num_occurrences** (*array_like, optional*) – Number of occurrences for each sample. If not provided, defaults to a vector of 1s.
- **aggregate_samples** (*bool, optional, default=False*) – If True, all samples in returned `SampleSet` are unique, with *num_occurrences* accounting for any duplicate samples in *samples_like*.
- **sort_labels** (*bool, optional, default=True*) – Return `SampleSet.variables` in sorted order. For mixed (unsortable) types, the given order is maintained.
- ****vectors** (*array_like*) – Other per-sample data.

Returns `SampleSet`

Examples

This example creates a SampleSet out of a samples_like object (a dict).

```
>>> import dimod
>>> import numpy as np
...
>>> dimod.SampleSet.from_samples(dimod.as_samples({'a': 0, 'b': 1, 'c': 0}),
...                              'BINARY', 0)    # doctest: +SKIP
SampleSet(rec.array([([0, 1, 0], 0, 1)],
...       dtype=[('sample', 'i1', (3,)), ('energy', '<i4'), ('num_occurrences', '
↪<i4')]),
...       ['a', 'b', 'c'], {}, 'BINARY')
```

dimod.SampleSet.from_samples_bqm

classmethod SampleSet.**from_samples_bqm**(*samples_like*, *bqm*, ***kwargs*)

Build a sample set from raw samples and a binary quadratic model.

The binary quadratic model is used to calculate energies and set the `vartype`.

Parameters

- **samples_like** – A collection of raw samples. 'samples_like' is an extension of NumPy's array_like. See `as_samples()`.
- **bqm** (`BinaryQuadraticModel`) – A binary quadratic model.
- **info** (*dict, optional*) – Information about the `SampleSet` as a whole formatted as a dict.
- **num_occurrences** (*array_like, optional*) – Number of occurrences for each sample. If not provided, defaults to a vector of 1s.

- **aggregate_samples** (*bool, optional, default=False*) – If True, all samples in returned `SampleSet` are unique, with *num_occurrences* accounting for any duplicate samples in *samples_like*.
- **sort_labels** (*bool, optional, default=True*) – Return `SampleSet.variables` in sorted order. For mixed (unsortable) types, the given order is maintained.
- ****vectors** (*array_like*) – Other per-sample data.

Returns `SampleSet`

Examples

```
>>> bqm = dimod.BinaryQuadraticModel.from_ising({}, {('a', 'b'): -1})
>>> samples = dimod.SampleSet.from_samples_bqm({'a': -1, 'b': 1}, bqm)
```

dimod.SampleSet.from_serializable

classmethod `SampleSet.from_serializable(obj)`

Deserialize a `SampleSet`.

Parameters **obj** (*dict*) – A `SampleSet` serialized by `to_serializable()`.

Returns `SampleSet`

Examples

This example encodes and decodes using JSON.

```
>>> import dimod
>>> import json
...
>>> samples = dimod.SampleSet.from_samples([-1, 1, -1], dimod.SPIN, energy=-.5)
>>> s = json.dumps(samples.to_serializable())
>>> new_samples = dimod.SampleSet.from_serializable(json.loads(s))
```

See also:

`to_serializable()`

dimod.SampleSet.lowest

`SampleSet.lowest(rtol=1e-05, atol=1e-08)`

Return a sample set containing the lowest-energy samples.

A sample is included if its energy is within tolerance of the lowest energy in the sample set. The following equation is used to determine if two values are equivalent:

absolute(a - b) <= ($atol$ + $rtol$ * absolute(b))

See `numpy.isclose()` for additional details and caveats.

Parameters

- **rtol** (*float, optional, default=1.e-5*) – The relative tolerance (see above).
- **atol** (*float, optional, default=1.e-8*) – The absolute tolerance (see above).

Returns A new sample set containing the lowest energy samples as delimited by configured tolerances from the lowest energy sample in the current sample set.

Return type *SampleSet*

Examples

```
>>> sampleset = dimod.ExactSolver().sample_ising({'a': .001},
...                                                {('a', 'b'): -1})
>>> print(sampleset.lowest())
   a  b energy num_oc.
0 -1 -1 -1.001       1
['SPIN', 1 rows, 1 samples, 2 variables]
>>> print(sampleset.lowest(atol=.1))
   a  b energy num_oc.
0 -1 -1 -1.001       1
1 +1 +1 -0.999       1
['SPIN', 2 rows, 2 samples, 2 variables]
```

Note: "Lowest energy" is the lowest energy in the sample set. This is not always the "ground energy" which is the lowest energy possible for a binary quadratic model.

dimod.SampleSet.resolve

SampleSet.**resolve**()

Ensure that the sampleset is resolved if constructed from a future.

dimod.SampleSet.relabel_variables

SampleSet.**relabel_variables**(*mapping*, *inplace=True*)

Relabel the variables of a *SampleSet* according to the specified mapping.

Parameters

- **mapping** (*dict*) – Mapping from current variable labels to new, as a dict. If incomplete mapping is specified, unmapped variables keep their current labels.
- **inplace** (*bool, optional, default=True*) – If True, the current *SampleSet* is updated; otherwise, a new *SampleSet* is returned.

Returns SampleSet with relabeled variables. If *inplace* is True, returns itself.

Return type *SampleSet*

Examples

This example creates a relabeled copy of a *SampleSet*.

```
>>> import dimod
...
>>> sampleset = dimod.ExactSolver().sample_ising({'a': -0.5, 'b': 1.0}, {('a', 'b
↪'): -1})
```

(continues on next page)

(continued from previous page)

```
>>> new_sampleset = sampleset.relabel_variables({'a': 0, 'b': 1}, inplace=False)
>>> sampleset.variable_labels    # doctest: +SKIP
[0, 1]
```

dimod.SampleSet.samples

SampleSet.**samples**(*n=None*, *sorted_by='energy'*)
Return an iterable over the samples.

Parameters

- **n** (*int, optional, default=None*) – Maximum number of samples to return in the view.
- **sorted_by** (*str/None, optional, default='energy'*) – Selects the record field used to sort the samples. If None, samples are returned in record order.

Returns A view object mapping variable labels to values.

Return type SamplesArray

Examples

```
>>> sampleset = dimod.ExactSolver().sample_ising({'a': 0.1, 'b': 0.0},
...                                               {('a', 'b'): 1})
>>> for sample in sampleset.samples():
...     print(sample)
{'a': -1, 'b': 1}
{'a': 1, 'b': -1}
{'a': -1, 'b': -1}
{'a': 1, 'b': 1}
```

```
>>> sampleset = dimod.ExactSolver().sample_ising({'a': 0.1, 'b': 0.0},
...                                               {('a', 'b'): 1})
>>> samples = sampleset.samples()
>>> samples[0]
{'a': -1, 'b': 1}
>>> samples[0, 'a']
-1
>>> samples[0, ['b', 'a']]
array([ 1, -1], dtype=int8)
>>> samples[1:, ['a', 'b']]
array([[ 1, -1],
       [-1, -1],
       [ 1,  1]], dtype=int8)
```

dimod.SampleSet.slice

SampleSet.**slice**(**slice_args*, ***kwargs*)
Create a new sample set with rows sliced according to standard Python slicing syntax.

Parameters

- **start** (*int, optional, default=None*) – Start index for *slice*.

- **stop** (*int*) – Stop index for *slice*.
- **step** (*int, optional, default=None*) – Step value for *slice*.
- **sorted_by** (*str/None, optional, default='energy'*) – Selects the record field used to sort the samples before slicing. Note that *sorted_by* determines the sample order in the returned sample set.

Returns *SampleSet*

Examples

```
>>> import numpy as np
...
>>> sampleset = dimod.SampleSet.from_samples(np.diag(range(1, 11)),
...                     dimod.BINARY, energy=range(10))
>>> print(sampleset)
   0  1  2  3  4  5  6  7  8  9 energy num_oc.
0  1  0  0  0  0  0  0  0  0  0      0       1
1  0  1  0  0  0  0  0  0  0  0      1       1
2  0  0  1  0  0  0  0  0  0  0      2       1
3  0  0  0  1  0  0  0  0  0  0      3       1
4  0  0  0  0  1  0  0  0  0  0      4       1
5  0  0  0  0  0  1  0  0  0  0      5       1
6  0  0  0  0  0  0  1  0  0  0      6       1
7  0  0  0  0  0  0  0  1  0  0      7       1
8  0  0  0  0  0  0  0  0  1  0      8       1
9  0  0  0  0  0  0  0  0  0  1      9       1
['BINARY', 10 rows, 10 samples, 10 variables]
```

The above example's first 3 samples by energy == truncate(3):

```
>>> print(sampleset.slice(3))
   0  1  2  3  4  5  6  7  8  9 energy num_oc.
0  1  0  0  0  0  0  0  0  0  0      0       1
1  0  1  0  0  0  0  0  0  0  0      1       1
2  0  0  1  0  0  0  0  0  0  0      2       1
['BINARY', 3 rows, 3 samples, 10 variables]
```

The last 3 samples by energy:

```
>>> print(sampleset.slice(-3, None))
   0  1  2  3  4  5  6  7  8  9 energy num_oc.
0  0  0  0  0  0  0  0  1  0  0      7       1
1  0  0  0  0  0  0  0  0  1  0      8       1
2  0  0  0  0  0  0  0  0  0  1      9       1
['BINARY', 3 rows, 3 samples, 10 variables]
```

Every second sample in between, skipping top and bottom 3:

```
>>> print(sampleset.slice(3, -3, 2))
   0  1  2  3  4  5  6  7  8  9 energy num_oc.
0  0  0  0  1  0  0  0  0  0  0      3       1
1  0  0  0  0  0  1  0  0  0  0      5       1
['BINARY', 2 rows, 2 samples, 10 variables]
```

dimod.SampleSet.to_pandas_dataframe

SampleSet.**to_pandas_dataframe**(*sample_column=False*)
Convert a sample set to a Pandas DataFrame

Parameters

- **sample_column** (*bool, optional, default=False*) – If True, samples are
- **as a column of type dict.** (*represented*) –

Returns pandas.DataFrame

Examples

```
>>> samples = dimod.SampleSet.from_samples([{'a': -1, 'b': +1, 'c': -1},
...                                          {'a': -1, 'b': -1, 'c': +1}],
...                                         dimod.SPIN, energy=-.5)
>>> samples.to_pandas_dataframe()   # doctest: +SKIP
   a  b  c  energy  num_occurrences
0 -1  1 -1    -0.5                1
1 -1 -1  1    -0.5                1
>>> samples.to_pandas_dataframe(sample_column=True)   # doctest: +SKIP
                       sample  energy  num_occurrences
0  {'a': -1, 'b': 1, 'c': -1}    -0.5                1
1  {'a': -1, 'b': -1, 'c': 1}    -0.5                1
```

dimod.SampleSet.to_serializable

SampleSet.**to_serializable**(*use_bytes=False, bytes_type=<class 'bytes'>*)
Convert a SampleSet to a serializable object.

Note that the contents of the SampleSet.info field are assumed to be serializable.

Parameters

- **use_bytes** (*bool, optional, default=False*) – If True, a compact representation of the biases as bytes is used.
- **bytes_type** (*class, optional, default=bytes*) – If *use_bytes* is True, this class is used to wrap the bytes objects in the serialization. Useful for Python 2 using BSON encoding, which does not accept the raw *bytes* type; *bson.Binary* can be used instead.

Returns Object that can be serialized.

Return type dict

Examples

This example encodes using JSON.

```
>>> import dimod
>>> import json
...
>>> samples = dimod.SampleSet.from_samples([-1, 1, -1], dimod.SPIN, energy=-.5)
>>> s = json.dumps(samples.to_serializable())
```

See also:

from_serializable()

dimod.SampleSet.truncate

`SampleSet.truncate(n, sorted_by='energy')`

Create a new sample set with up to n rows.

Parameters

- **n** (*int*) – Maximum number of rows in the returned sample set. Does not return any rows above this limit in the original sample set.
- **sorted_by** (*str/None, optional, default='energy'*) – Selects the record field used to sort the samples before truncating. Note that this sort order is maintained in the returned sample set.

Returns *SampleSet*

Examples

```
>>> import numpy as np
...
>>> sampleset = dimod.SampleSet.from_samples(np.ones((5, 5)), dimod.SPIN,␣
↪energy=5)
>>> print(sampleset)
   0  1  2  3  4 energy num_oc.
0 +1 +1 +1 +1 +1      5       1
1 +1 +1 +1 +1 +1      5       1
2 +1 +1 +1 +1 +1      5       1
3 +1 +1 +1 +1 +1      5       1
4 +1 +1 +1 +1 +1      5       1
['SPIN', 5 rows, 5 samples, 5 variables]
>>> print(sampleset.truncate(2))
   0  1  2  3  4 energy num_oc.
0 +1 +1 +1 +1 +1      5       1
1 +1 +1 +1 +1 +1      5       1
['SPIN', 2 rows, 2 samples, 5 variables]
```

See: *SampleSet.slice()*

Utility Functions

concatenate(samplesets[, defaults])	Combine sample sets.

dimod.concatenate

`concatenate(samplesets, defaults=None)`

Combine sample sets.

Parameters

- **samplesets** (iterable[*SampleSet*]) – Iterable of sample sets.

- **defaults** (*dict, optional*) – Dictionary mapping data vector names to the corresponding default values.

Returns A sample set with the same vartype and variable order as the first given in *samplesets*.

Return type *SampleSet*

Examples

```
>>> a = dimod.SampleSet.from_samples(([-1, +1], 'ab'), dimod.SPIN, energy=-1)
>>> b = dimod.SampleSet.from_samples(([-1, +1], 'ba'), dimod.SPIN, energy=-1)
>>> ab = dimod.concatenate((a, b))
>>> ab.record.sample
array([[-1,  1],
       [ 1, -1]], dtype=int8)
```

2.2.4 Higher-Order Models

Sometimes it is nice to work with problems that are not restricted to quadratic interactions.

Binary Polynomials

class BinaryPolynomial(*poly*, *vartype*)

A polynomial with binary variables and real-valued coefficients.

Parameters

- **poly** (*mapping/iterable*) – Polynomial as a mapping of form {term: bias, ...}, where *term* is a collection of variables and *bias* the associated bias. It can also be an iterable of 2-tuples (term, bias).
- **vartype** (*Vartype*/str/set) – Variable type for the binary quadratic model. Accepted input values:
  - `Vartype.SPIN`, `'SPIN'`, `{-1, 1}`
  - `Vartype.BINARY`, `'BINARY'`, `{0, 1}`

degree

The degree of the polynomial.

Type int

variables

The variables.

Type set

vartype

One of `Vartype.SPIN` or `Vartype.BINARY`.

Type *Vartype*

Examples

Binary polynomials can be constructed in many different ways. The following are all equivalent

```
>>> poly = dimod.BinaryPolynomial({'a': -1, 'ab': 1}, dimod.SPIN)
>>> poly = dimod.BinaryPolynomial({('a',): -1, ('a', 'b'): 1}, dimod.SPIN)
>>> poly = dimod.BinaryPolynomial([('a', -1), (('a', 'b'), 1)], dimod.SPIN)
>>> poly = dimod.BinaryPolynomial({'a': -1, 'ab': .5, 'ba': .5}, dimod.SPIN)
```

Binary polynomials act a mutable mappings but the terms can be accessed with any sequence.

```
>>> poly = dimod.BinaryPolynomial({'a': -1, 'ab': 1}, dimod.BINARY)
>>> poly['ab']
1
>>> poly['ba']
1
>>> poly[{'a', 'b'}]
1
>>> poly[('a', 'b')]
1
>>> poly['cd'] = 4
>>> poly['dc']
4
```

Methods

BinaryPolynomial.copy()	Create a shallow copy.
BinaryPolynomial.energies(samples_like[, dtype])	The energies of the given samples.
BinaryPolynomial.energy(sample_like[, dtype])	The energy of the given sample.
BinaryPolynomial.from_hising(h, J[, offset])	Construct a binary polynomial from a higher-order Ising problem.
BinaryPolynomial.from_hubo(H[, offset])	Construct a binary polynomial from a higher-order unconstrained binary optimization (HUBO) problem.
BinaryPolynomial.normalize([bias_range, ...])	Normalizes the biases of the binary polynomial such that they fall in the provided range(s).
BinaryPolynomial.relabel_variables(mapping)	Relabel variables of a binary polynomial as specified by mapping.
BinaryPolynomial.scale(scalar[, ignored_terms])	Multiply the polynomial by the given scalar.
BinaryPolynomial.to_binary([copy])	Return a binary polynomial over *{0, 1}* variables.
BinaryPolynomial.to_hising()	Construct a higher-order Ising problem from a binary polynomial.
BinaryPolynomial.to_hubo()	Construct a higher-order unconstrained binary optimization (HUBO) problem from a binary polynomial.
BinaryPolynomial.to_spin([copy])	Return a binary polynomial over *{-1, +1}* variables.

dimod.higherorder.polynomial.BinaryPolynomial.copy

BinaryPolynomial.**copy**()

Create a shallow copy.

dimod.higherorder.polynomial.BinaryPolynomial.energies

BinaryPolynomial.**energies**(*samples_like*, *dtype=<class 'float'>*)

The energies of the given samples.

Parameters

- **samples_like** (*samples_like*) – A collection of raw samples. *samples_like* is an extension of NumPy's array_like structure. See `as_samples()`.
- **dtype** (`numpy.dtype`, optional) – The data type of the returned energies. Defaults to float.

Returns The energies.

Return type `numpy.ndarray`

dimod.higherorder.polynomial.BinaryPolynomial.energy

BinaryPolynomial.**energy**(*sample_like*, *dtype=<class 'float'>*)

The energy of the given sample.

Parameters

- **sample_like** (*samples_like*) – A raw sample. *sample_like* is an extension of NumPy's array_like structure. See `as_samples()`.
- **dtype** (`numpy.dtype`, optional) – The data type of the returned energies. Defaults to float.

Returns The energy.

dimod.higherorder.polynomial.BinaryPolynomial.from_hising

classmethod BinaryPolynomial.**from_hising**(*h*, *J*, *offset=None*)

Construct a binary polynomial from a higher-order Ising problem.

Parameters

- **h** (*dict*) – The linear biases.
- **J** (*dict*) – The higher-order biases.
- **offset** (*optional, default=0.0*) – Constant offset applied to the model.

Returns `BinaryPolynomial`

Examples

```
>>> poly = dimod.BinaryPolynomial.from_hising({'a': 2}, {'ab': -1}, 0)
```

dimod.higherorder.polynomial.BinaryPolynomial.from_hubo

classmethod BinaryPolynomial.**from_hubo**(*H*, *offset=None*)

Construct a binary polynomial from a higher-order unconstrained binary optimization (HUBO) problem.

Parameters **H** (*dict*) – Coefficients of a higher-order unconstrained binary optimization (HUBO) model.

Returns *BinaryPolynomial*

Examples

```
>>> poly = dimod.BinaryPolynomial.from_hubo({('a', 'b', 'c'): -1})
```

dimod.higherorder.polynomial.BinaryPolynomial.normalize

BinaryPolynomial.**normalize**(*bias_range=1*, *poly_range=None*, *ignored_terms=None*)

Normalizes the biases of the binary polynomial such that they fall in the provided range(s).

If *poly_range* is provided, then *bias_range* will be treated as the range for the linear biases and *poly_range* will be used for the range of the other biases.

Parameters

- **bias_range** (*number/pair*) – Value/range by which to normalize the all the biases, or if *poly_range* is provided, just the linear biases.
- **poly_range** (*number/pair, optional*) – Value/range by which to normalize the higher order biases.
- **ignored_terms** (*iterable, optional*) – Biases associated with these terms are not scaled.

dimod.higherorder.polynomial.BinaryPolynomial.relabel_variables

BinaryPolynomial.**relabel_variables**(*mapping*, *inplace=True*)

Relabel variables of a binary polynomial as specified by mapping.

Parameters

- **mapping** (*dict*) – Dict mapping current variable labels to new ones. If an incomplete mapping is provided, unmapped variables retain their current labels.
- **inplace** (*bool, optional, default=True*) – If True, the binary polynomial is updated in-place; otherwise, a new binary polynomial is returned.

Returns A binary polynomial with the variables relabeled. If *inplace* is set to True, returns itself.

Return type *BinaryPolynomial*

dimod.higherorder.polynomial.BinaryPolynomial.scale

BinaryPolynomial.**scale**(*scalar*, *ignored_terms=None*)

Multiply the polynomial by the given scalar.

Parameters

- **scalar** (*number*) – Value to multiply the polynomial by.
- **ignored_terms** (*iterable, optional*) – Biases associated with these terms are not scaled.

dimod.higherorder.polynomial.BinaryPolynomial.to_binary

`BinaryPolynomial.to_binary(copy=False)`

Return a binary polynomial over *{0, 1}* variables.

Parameters **copy** (*optional, default=False*) – If True, the returned polynomial is always a copy. Otherwise, if the polynomial is binary-valued already it returns itself.

Returns *BinaryPolynomial*

dimod.higherorder.polynomial.BinaryPolynomial.to_hising

`BinaryPolynomial.to_hising()`

Construct a higher-order Ising problem from a binary polynomial.

Returns A 3-tuple of the form (*h*, *J*, *offset*) where *h* includes the linear biases, *J* has the higher-order biases and *offset* is the linear offset.

Return type tuple

Examples

```
>>> poly = dimod.BinaryPolynomial({'a': -1, 'ab': 1, 'abc': -1}, dimod.SPIN)
>>> h, J, off = poly.to_hising()
>>> h
{'a': -1}
```

dimod.higherorder.polynomial.BinaryPolynomial.to_hubo

`BinaryPolynomial.to_hubo()`

Construct a higher-order unconstrained binary optimization (HUBO) problem from a binary polynomial.

Returns A 2-tuple of the form (*H*, *offset*) where *H* is the HUBO and *offset* is the linear offset.

Return type tuple

dimod.higherorder.polynomial.BinaryPolynomial.to_spin

`BinaryPolynomial.to_spin(copy=False)`

Return a binary polynomial over *{-1, +1}* variables.

Parameters **copy** (*optional, default=False*) – If True, the returned polynomial is always a copy. Otherwise, if the polynomial is spin-valued already it returns itself.

Returns *BinaryPolynomial*

Reducing to a Binary Quadratic Model

make_quadratic(poly, strength[, vartype, bqm])	Create a binary quadratic model from a higher order polynomial.

dimod.higherorder.utils.make_quadratic

make_quadratic(*poly*, *strength*, *vartype=None*, *bqm=None*)

Create a binary quadratic model from a higher order polynomial.

Parameters

- **poly** (*dict*) – Polynomial as a dict of form {term: bias, …}, where *term* is a tuple of variables and *bias* the associated bias.
- **strength** (*float*) – The energy penalty for violating the prodcut constraint. Insufficient strength can result in the binary quadratic model not having the same minimizations as the polynomial.
- **vartype** (*Vartype*/str/set, optional) – Variable type for the binary quadratic model. Accepted input values:
  - `Vartype.SPIN`, `'SPIN'`, `{-1, 1}`
  - `Vartype.BINARY`, `'BINARY'`, `{0, 1}`

  If *bqm* is provided, *vartype* is not required.
- **bqm** (*BinaryQuadraticModel*, optional) – The terms of the reduced polynomial are added to this binary quadratic model. If not provided, a new binary quadratic model is created.

Returns *BinaryQuadraticModel*

Examples

```
>>> poly = {(0,): -1, (1,): 1, (2,): 1.5, (0, 1): -1, (0, 1, 2): -2}
>>> bqm = dimod.make_quadratic(poly, 5.0, dimod.SPIN)
```

2.2.5 Utilities

Contents

Energy Calculations

ising_energy(sample, h, J[, offset])	Calculate the energy for the specified sample of an Ising model.
qubo_energy(sample, Q[, offset])	Calculate the energy for the specified sample of a QUBO model.

dimod.utilities.ising_energy

ising_energy (*sample*, *h*, *J*, *offset=0.0*)

Calculate the energy for the specified sample of an Ising model.

Energy of a sample for a binary quadratic model is defined as a sum, offset by the constant energy offset associated with the model, of the sample multipled by the linear bias of the variable and all its interactions. For an Ising model,

$$E(\mathbf{s}) = \sum_v h_v s_v + \sum_{u,v} J_{u,v} s_u s_v + c$$

where s_v is the sample, h_v is the linear bias, $J_{u,v}$ the quadratic bias (interactions), and c the energy offset.

Parameters

- **sample** (*dict[variable, spin]*) – Sample for a binary quadratic model as a dict of form {v: spin, . . . }, where keys are variables of the model and values are spins (either -1 or 1).
- **h** (*dict[variable, bias]*) – Linear biases as a dict of the form {v: bias, . . . }, where keys are variables of the model and values are biases.
- **J** (*dict[(variable, variable), bias]*) – Quadratic biases as a dict of the form {(u, v): bias, . . . }, where keys are 2-tuples of variables of the model and values are quadratic biases associated with the pair of variables (the interaction).
- **offset** (*numeric, optional, default=0*) – Constant offset to be applied to the energy. Default 0.

Returns The induced energy.

Return type float

Notes

No input checking is performed.

Examples

This example calculates the energy of a sample representing two down spins for an Ising model of two variables that have positive biases of value 1 and are positively coupled with an interaction of value 1.

```
>>> import dimod
>>> sample = {1: -1, 2: -1}
>>> h = {1: 1, 2: 1}
>>> J = {(1, 2): 1}
>>> dimod.ising_energy(sample, h, J, 0.5)
-0.5
```

References

Ising model on Wikipedia

dimod.utilities.qubo_energy

qubo_energy (*sample*, *Q*, *offset=0.0*)

Calculate the energy for the specified sample of a QUBO model.

Energy of a sample for a binary quadratic model is defined as a sum, offset by the constant energy offset associated with the model, of the sample multipled by the linear bias of the variable and all its interactions. For a quadratic unconstrained binary optimization (QUBO) model,

$$E(\mathbf{x}) = \sum_{u,v} Q_{u,v} x_u x_v + c$$

where x_v is the sample, $Q_{u,v}$ a matrix of biases, and c the energy offset.

Parameters

- **sample** (*dict[variable, spin]*) – Sample for a binary quadratic model as a dict of form {v: bin, . . . }, where keys are variables of the model and values are binary (either 0 or 1).
- **Q** (*dict[(variable, variable), coefficient]*) – QUBO coefficients in a dict of form {(u, v): coefficient, . . . }, where keys are 2-tuples of variables of the model and values are biases associated with the pair of variables. Tuples (u, v) represent interactions and (v, v) linear biases.
- **offset** (*numeric, optional, default=0*) – Constant offset to be applied to the energy. Default 0.

Returns The induced energy.

Return type float

Notes

No input checking is performed.

Examples

This example calculates the energy of a sample representing two zeros for a QUBO model of two variables that have positive biases of value 1 and are positively coupled with an interaction of value 1.

```
>>> import dimod
>>> sample = {1: 0, 2: 0}
>>> Q = {(1, 1): 1, (2, 2): 1, (1, 2): 1}
>>> dimod.qubo_energy(sample, Q, 0.5)
0.5
```

References

QUBO model on Wikipedia

Decorators

Decorators can be imported from the `dimod.decorators` namespace. For example:

```
>>> from dimod.decorators import vartype_argument
```

`bqm_index_labels`(f)	Decorator to convert a BQM to index-labels and relabel the sample set output.
`bqm_index_labelled_input`(…)	Returns a decorator that ensures BQM variable labeling and specified sample_like inputs are index labeled and consistent.
`bqm_structured`(f)	Decorator to raise an error if the given BQM does not match the sampler's structure.
`graph_argument`(*arg_names, **options)	Decorator to coerce given graph arguments into a consistent form.
`vartype_argument`(*arg_names)	Ensures the wrapped function receives valid vartype argument(s).

dimod.decorators.bqm_index_labels

bqm_index_labels(*f*)

Decorator to convert a BQM to index-labels and relabel the sample set output.

Designed to be applied to `Sampler.sample()`. Expects the wrapped function or method to accept a `BinaryQuadraticModel` as the second input and to return a `SampleSet`.

dimod.decorators.bqm_index_labelled_input

bqm_index_labelled_input(*var_labels_arg_name*, *samples_arg_names*)

Returns a decorator that ensures BQM variable labeling and specified sample_like inputs are index labeled and consistent.

Parameters

- **var_labels_arg_name** (`str`) – Expected name of the argument used to pass in an index labeling for the binary quadratic model (BQM).
- **samples_arg_names** (`list[str]`) – Expected names of sample_like inputs that should be indexed by the labels passed to the *var_labels_arg_name* argument. 'samples_like' is an extension of NumPy's array_like. See `as_samples()`.

Returns Function decorator.

dimod.decorators.bqm_structured

bqm_structured(*f*)

Decorator to raise an error if the given BQM does not match the sampler's structure.

Designed to be applied to `Sampler.sample()`. Expects the wrapped function or method to accept a `BinaryQuadraticModel` as the second input and for the `Sampler` to also be `Structured`.

dimod.decorators.graph_argument

graph_argument (**arg_names*, ***options*)

Decorator to coerce given graph arguments into a consistent form.

The wrapped function accepts either an integer n, interpreted as a complete graph of size n, a nodes/edges pair, a sequence of edges, or a NetworkX graph. The argument is converted into a nodes/edges 2-tuple.

Parameters

- ***arg_names** (*optional, default='G'*) – Names of the arguments for input graphs.
- **allow_None** (*bool, optional, default=False*) – If True, None can be passed through as an input graph.

dimod.decorators.vartype_argument

vartype_argument (**arg_names*)

Ensures the wrapped function receives valid vartype argument(s).

One or more argument names can be specified as a list of string arguments.

Parameters ***arg_names** (*list[str], argument names, optional, default='vartype'*) – Names of the constrained arguments in decorated function.

Returns Function decorator.

Examples

```
>>> from dimod.decorators import vartype_argument
```

```
>>> @vartype_argument()
... def f(x, vartype):
...     print(vartype)
...
>>> f(1, 'SPIN')
Vartype.SPIN
>>> f(1, vartype='SPIN')
Vartype.SPIN
```

```
>>> @vartype_argument('y')
... def f(x, y):
...     print(y)
...
>>> f(1, 'SPIN')
Vartype.SPIN
>>> f(1, y='SPIN')
Vartype.SPIN
```

```
>>> @vartype_argument('z')
... def f(x, **kwargs):
...     print(kwargs['z'])
...
>>> f(1, z='SPIN')
Vartype.SPIN
```

Note: The decorated function can explicitly list (name) vartype arguments constrained by `vartype_argument()` or it can use a keyword arguments *dict*.

See also:

`as_vartype()`

Graph-like

`child_structure_dfs`(sampler[, seen])	Return the structure of a composed sampler using a depth-first search on its children.

dimod.utilities.child_structure_dfs

child_structure_dfs(*sampler*, *seen=None*)

Return the structure of a composed sampler using a depth-first search on its children.

Parameters

- **sampler** (`Sampler`) – `Structured` or composed sampler with at least one structured child.
- **seen** (`set, optional, default=False`) – IDs of already checked child samplers.

Returns A named tuple of the form *Structure(nodelist, edgelist, adjacency)*, where the 3-tuple values are the `Structured.nodelist`, `Structured.edgelist` and `Structured.adjacency` attributes of the first structured sampler found.

Return type `namedtuple`

Raises `ValueError` – If no structured sampler is found.

Examples:

```
>>> import dimod
>>> sampler = dimod.TrackingComposite(
...             dimod.StructureComposite(
...             dimod.ExactSolver(), [0, 1], [(0, 1)]))
>>> print(dimod.child_structure_dfs(sampler).nodelist)
[0, 1]
```

Serialization

JSON

JSON-encoding of dimod objects.

Examples

```
>>> import json
>>> from dimod.serialization.json import DimodEncoder, DimodDecoder
...
```

(continues on next page)

(continued from previous page)

```
>>> bqm = dimod.BinaryQuadraticModel.from_ising({}, {('a', 'b'): -1})
>>> s = json.dumps(bqm, cls=DimodEncoder)
>>> new = json.loads(s, cls=DimodDecoder)
>>> bqm == new
True
```

```
>>> import json
>>> from dimod.serialization.json import DimodEncoder, DimodDecoder
...
>>> sampleset = dimod.SampleSet.from_samples({'a': -1, 'b': 1}, dimod.SPIN, energy=5)
>>> s = json.dumps(sampleset, cls=DimodEncoder)
>>> new = json.loads(s, cls=DimodDecoder)
>>> sampleset == new
True
```

```
>>> import json
>>> from dimod.serialization.json import DimodEncoder, DimodDecoder
...
>>> # now inside a list
>>> s = json.dumps([sampleset, bqm], cls=DimodEncoder)
>>> new = json.loads(s, cls=DimodDecoder)
>>> new == [sampleset, bqm]
True
```

class DimodEncoder(**, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, sort_keys=False, indent=None, separators=None, default=None*)

Subclass the JSONEncoder for dimod objects.

class DimodDecoder(**args, **kwargs*)

Subclass the JSONDecoder for dimod objects.

Uses `dimod_object_hook()`.

Functions

`dimod_object_hook`(obj)	JSON-decoding for dimod objects.

dimod.serialization.json.dimod_object_hook

dimod_object_hook(*obj*)

JSON-decoding for dimod objects.

See also:

`json.JSONDecoder` for using custom decoders.

Testing

The testing subpackage contains functions for verifying and testing dimod objects. Testing objects/functions can be imported from the `dimod.testing` namespace. For example:

```
>>> from dimod.testing import assert_sampler_api
```

API Asserts

assert_composite_api(composed_sampler)	Assert that an instantiated composed sampler exposes correct composite properties and methods.
assert_sampler_api(sampler)	Assert that an instantiated sampler exposes correct properties and methods.
assert_structured_api(sampler)	Assert that an instantiated structured sampler exposes correct composite properties and methods.

dimod.testing.asserts.assert_composite_api

assert_composite_api(*composed_sampler*)

Assert that an instantiated composed sampler exposes correct composite properties and methods.

Parameters **sampler** (*Composite*) – User-made dimod composed sampler.

Raises AssertionError – If the given sampler does not match the composite API.

See also:

Composite for the abstract base class that defines the composite API.

assert_sampler_api to assert that the composed sampler matches the sampler API.

dimod.testing.asserts.assert_sampler_api

assert_sampler_api(*sampler*)

Assert that an instantiated sampler exposes correct properties and methods.

Parameters **sampler** (*Sampler*) – User-made dimod sampler.

Raises AssertionError – If the given sampler does not match the sampler API.

See also:

Sampler for the abstract base class that defines the sampler API.

dimod.testing.asserts.assert_structured_api

assert_structured_api(*sampler*)

Assert that an instantiated structured sampler exposes correct composite properties and methods.

Parameters **sampler** (*Structured*) – User-made dimod structured sampler.

Raises AssertionError – If the given sampler does not match the structured API.

See also:

Structured for the abstract base class that defines the structured API.

assert_sampler_api to assert that the structured sampler matches the sampler API.

Correctness Asserts

assert_bqm_almost_equal(actual, desired[, ...])	Test if two binary quadratic models have almost equal biases.
assert_response_energies(response, bqm[, ...])	Assert that each sample in the given response has the correct energy.
assert_sampleset_energies(sampleset, bqm[, ...])	Assert that each sample in the given sample set has the correct energy.

dimod.testing.asserts.assert_bqm_almost_equal

assert_bqm_almost_equal(*actual*, *desired*, *places=7*, *ignore_zero_interactions=False*)
Test if two binary quadratic models have almost equal biases.

Parameters

- **actual** (*BinaryQuadraticModel*) – First binary quadratic model.
- **desired** (*BinaryQuadraticModel*) – Second binary quadratic model.
- **places** (*int, optional, default=7*) – Bias equality is computed as `round(b0 - b1, places) == 0`.
- **ignore_zero_interactions** (*bool, optional, default=False*) – If true, interactions with 0 bias are ignored.

dimod.testing.asserts.assert_response_energies

assert_response_energies(*response*, *bqm*, *precision=7*)
Assert that each sample in the given response has the correct energy.

Parameters

- **response** (*SampleSet*) – Response as returned by a dimod sampler.
- **bqm** (*BinaryQuadraticModel*) – Binary quadratic model (BQM) used to generate the samples.
- **precision** (*int, optional, default=7*) – Equality of energy is tested by calculating the difference between the *response*'s sample energy and that returned by BQM's *energy()*, rounding to the closest multiple of 10 to the power of minus *precision*.

Raises

- AssertionError – If any of the samples in the response do not match their
- associated energy.

See also:

assert_sampleset_energies()

dimod.testing.asserts.assert_sampleset_energies

assert_sampleset_energies(*sampleset*, *bqm*, *precision=7*)
Assert that each sample in the given sample set has the correct energy.

Parameters

- **sampleset** (*SampleSet*) – Sample set as returned by a dimod sampler.

- **bqm** (*BinaryQuadraticModel*/*BinaryPolynomial*) – The binary quadratic model (BQM) or binary polynomial used to generate the samples.
- **precision** (*int, optional, default=7*) – Equality of energy is tested by calculating the difference between the *response*'s sample energy and that returned by BQM's `energy()`, rounding to the closest multiple of 10 to the power of minus *precision*.

Raises

- `AssertionError` – If any of the samples in the sample set do not match
- their associated energy.

Examples

```
>>> import dimod.testing
...
>>> sampler = dimod.ExactSolver()
>>> bqm = dimod.BinaryQuadraticModel.from_ising({}, {(0, 1): -1})
>>> sampleset = sampler.sample(bqm)
>>> dimod.testing.assert_response_energies(sampleset, bqm)
```

Vartype Conversion

ising_to_qubo(h, J[, offset])	Convert an Ising problem to a QUBO problem.
qubo_to_ising(Q[, offset])	Convert a QUBO problem to an Ising problem.

dimod.utilities.ising_to_qubo

ising_to_qubo(*h*, *J*, *offset=0.0*)

Convert an Ising problem to a QUBO problem.

Map an Ising model defined on spins (variables with {-1, +1} values) to quadratic unconstrained binary optimization (QUBO) formulation $x'Qx$ defined over binary variables (0 or 1 values), where the linear term is contained along the diagonal of Q. Return matrix Q that defines the model as well as the offset in energy between the two problem formulations:

$$s'Js + h's = offset + x'Qx$$

See `qubo_to_ising()` for the inverse function.

Parameters

- **h** (*dict[variable, bias]*) – Linear biases as a dict of the form {v: bias, . . . }, where keys are variables of the model and values are biases.
- **J** (*dict[(variable, variable), bias]*) – Quadratic biases as a dict of the form {(u, v): bias, . . . }, where keys are 2-tuples of variables of the model and values are quadratic biases associated with the pair of variables (the interaction).
- **offset** (*numeric, optional, default=0*) – Constant offset to be applied to the energy. Default 0.

Returns

A 2-tuple containing:

dict: QUBO coefficients.

float: New energy offset.

Return type (dict, float)

Examples

This example converts an Ising problem of two variables that have positive biases of value 1 and are positively coupled with an interaction of value 1 to a QUBO problem.

```
>>> import dimod
>>> h = {1: 1, 2: 1}
>>> J = {(1, 2): 1}
>>> dimod.ising_to_qubo(h, J, 0.5)  # doctest: +SKIP
({(1, 1): 0.0, (1, 2): 4.0, (2, 2): 0.0}, -0.5)
```

dimod.utilities.qubo_to_ising

qubo_to_ising(*Q*, *offset=0.0*)

Convert a QUBO problem to an Ising problem.

Map a quadratic unconstrained binary optimization (QUBO) problem $x'Qx$ defined over binary variables (0 or 1 values), where the linear term is contained along the diagonal of Q, to an Ising model defined on spins (variables with {-1, +1} values). Return h and J that define the Ising model as well as the offset in energy between the two problem formulations:

$$x'Qx = offset + s'Js + h's$$

See `ising_to_qubo()` for the inverse function.

Parameters

- **Q** (*dict[(variable, variable), coefficient]*) – QUBO coefficients in a dict of form {(u, v): coefficient, . . . }, where keys are 2-tuples of variables of the model and values are biases associated with the pair of variables. Tuples (u, v) represent interactions and (v, v) linear biases.
- **offset** (*numeric, optional, default=0*) – Constant offset to be applied to the energy. Default 0.

Returns

A 3-tuple containing:

dict: Linear coefficients of the Ising problem.

dict: Quadratic coefficients of the Ising problem.

float: New energy offset.

Return type (dict, dict, float)

Examples

This example converts a QUBO problem of two variables that have positive biases of value 1 and are positively coupled with an interaction of value 1 to an Ising problem.

```
>>> import dimod
>>> Q = {(1, 1): 1, (2, 2): 1, (1, 2): 1}
>>> dimod.qubo_to_ising(Q, 0.5)     # doctest: +SKIP
({1: 0.75, 2: 0.75}, {(1, 2): 0.25}, 1.75)
```

2.2.6 Vartype

Enumeration of valid variable types for binary quadratic models.

Examples

Vartype is an Enum. Each vartype has a value and a name.

```
>>> vartype = dimod.SPIN
>>> vartype.name
'SPIN'
>>> vartype.value == {-1, +1}
True
```

```
>>> vartype = dimod.BINARY
>>> vartype.name
'BINARY'
>>> vartype.value == {0, 1}
True
```

The *as_vartype()* function allows the user to provide several convenient forms.

```
>>> from dimod import as_vartype
```

```
>>> as_vartype(dimod.SPIN) is dimod.SPIN
True
>>> as_vartype('SPIN') is dimod.SPIN
True
>>> as_vartype({-1, 1}) is dimod.SPIN
True
```

```
>>> as_vartype(dimod.BINARY) is dimod.BINARY
True
>>> as_vartype('BINARY') is dimod.BINARY
True
>>> as_vartype({0, 1}) is dimod.BINARY
True
```

class Vartype

An Enum over the types of variables for the binary quadratic model.

SPIN

Vartype for spin-valued models; variables of the model are either -1 or 1.

Type *Vartype*

BINARY

Vartype for binary models; variables of the model are either 0 or 1.

Type *Vartype*

as_vartype (*vartype*)
Cast various inputs to a valid vartype object.

Parameters **vartype** (*Vartype*/str/set) – Variable type. Accepted input values:

- `Vartype.SPIN`, `'SPIN'`, `{-1, 1}`
- `Vartype.BINARY`, `'BINARY'`, `{0, 1}`

Returns Either `Vartype.SPIN` or `Vartype.BINARY`.

Return type *Vartype*

See also:

vartype_argument()

2.2.7 Exceptions

exception BinaryQuadraticModelSizeError
Raised when the binary quadratic model has too many variables

exception BinaryQuadraticModelStructureError
Raised when the binary quadratic model does not fit the sampler

exception BinaryQuadraticModelValueError
Raised when a sampler cannot handle a specified binary quadratic model

exception InvalidComposition
Raised for compositions of samplers that are invalid

exception InvalidSampler
Raised when trying to use the specified sampler as a sampler

exception MappingError
Raised when mapping causes conflicting values in samples

exception WriteableError
Raised when trying to modify an immutable object.

2.3 Bibliography

2.4 Installation

Compatible with Python 2 and 3:

```
pip install dimod
```

To install with optional components:

```
pip install dimod[all]
```

To install from source:

```
pip install -r requirements.txt
python setup.py install
```

Note that for an installation from source some functionality requires that your system have Boost C++ libraries installed.

2.5 License

Apache License

Version 2.0, January 2004

http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

   "Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

   "Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

   "Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

   "Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

   "Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

   "Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

   (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

   You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without

limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

> To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets "[]" replaced with your own identifying information. (Don't include the brackets!) The text should be enclosed in the appropriate comment syntax for the file format. We also recommend that a file or class name and description of purpose be included on the same "printed page" as the copyright notice for easier identification within third-party archives.

Copyright [yyyy] [name of copyright owner]

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

> http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search
- Glossary

Bibliography

[Kin2015] James King, Sheir Yarkoni, Mayssam M. Nevisi, Jeremy P. Hilton, Catherine C. McGeoch. Benchmarking a quantum annealing processor with the time-to-target metric. https://arxiv.org/abs/1508.05087

[BHT] Boros, E., P.L. Hammer, G. Tavares. Preprocessing of Unconstraint Quadratic Binary Optimization. Rutcor Research Report 10-2006, April, 2006.

[BH] Boros, E., P.L. Hammer. Pseudo-Boolean optimization. Discrete Applied Mathematics 123, (2002), pp. 155-225

Python Module Index

Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W