QuiverTools

Release 1.0

Pieter Belmans, Hans Franzen, Gianni Petrella

Jul 04, 2024

CONTENTS

QuiverTools is a SageMath package to deal with quivers and moduli of quiver representations. Below you can find its documentation. A more detailed user guide is in the works.

To install it, run

```
sage --pip install git+https://github.com/QuiverTools/QuiverTools.git
```

and then you can simply run

```
from quiver import *
```

to get started.

For more information, see https://quiver.tools.

Authors

- Pieter Belmans (University of Luxembourg)
- Hans Franzen (University of Paderborn)
- Gianni Petrella (University of Luxembourg)

Funding

We acknowledge the generous support of:

- the Luxembourg National Research Fund (FNR–17113194 and FNR–17953441)
- the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) SFB-TRR 358/1 2023 “Integral Structures in Geometry and Representation Theory” (491392403)

CHAPTER

ONE

QUIVERS

class quiver.**Quiver**(*G*, *name=None*)

A quiver is a (finite) directed multigraph. It is an important tool in the representation theory of (finite-dimensional) algebras, because it allows one to construct the path algebra, whose modules are equivalently described as representations of the quiver. These in turn can be classified using moduli spaces of quiver representations.

For an introduction to the subject one is referred to

- Harm Derksen and Jerzy Weyman: An introduction to quiver representations
- Markus Reineke: Moduli of representations of quivers

or one of the many other resources that exist.

__init__(*G*, *name=None*)

Constructor for a quiver.

This takes a directed graph as input. If it is not a *DiGraph* instance, it is interpreted it as an adjacency matrix. For other constructions, see

- *Quiver.from_digraph()*
- *Quiver.from_matrix()*
- *Quiver.from_string()*

INPUT:

- `G` – directed graph
- `name` – optional name for the quiver

EXAMPLES:

The 3-Kronecker quiver from an adjacency matrix:

```
sage: from quiver import *
sage: Q = Quiver([[0, 3], [0, 0]]); Q
a quiver with 2 vertices and 3 arrows
```

classmethod from_digraph(*G*, *name=None*)

Construct a quiver from a *DiGraph* object.

INPUT:

- `G` – directed graph as a *DiGraph* object
- `name` – optional name for the quiver

OUTPUT: the quiver.

EXAMPLES:

The 3-Kronecker quiver:

```
sage: from quiver import *
sage: M = [[0, 3], [0, 0]]
sage: Quiver.from_digraph(DiGraph(matrix(M))) == Quiver.from_matrix(M)
True
```

classmethod from_matrix(*M*, *name=None*)

Construct a quiver from its adjacency matrix.

INPUT:

- `M` – adjacency matrix of the quiver
- `name` – optional name for the quiver

OUTPUT: the quiver.

EXAMPLES:

The 3-Kronecker quiver:

```
sage: from quiver import *
sage: Q = Quiver.from_matrix([[0, 3], [0, 0]]); Q.adjacency_matrix()
[0 3]
[0 0]
```

classmethod from_string(*Q: str*, *forget_labels=True*, *name=None*)

Construct a quiver from a comma-separated list of chains like `i-j-k-...`

You specify an arrow from `i` to `j` by writing `i-j`. Multiple arrows are specified by repeating the hyphen, so that `1--2` is the Kronecker quiver. If you write `i-j-k` then you have 1 arrow from `i` to `j` and one from `j` to `k`. The full quiver is specified by concatenating (multiple) arrows by commas.

The values for a vertex can be anything, and the chosen names will be used for the vertices in the underlying graph. Labels are cast to an integer, if possible, and otherwise to strings.

INPUT:

- `Q` – a string of the format described above giving a quiver
- `forget_labels` – (default: True): whether to use labels for vertices or to number them `0,...,n-1`
- `name` – optional name for the quiver

OUTPUT: the quiver

EXAMPLES:

The 3-Kronecker quiver defined in two different ways:

```
sage: from quiver import *
sage: Quiver.from_matrix([[0, 3], [0, 0]]) == Quiver.from_string("a---b")
True
```

A more complicated example:

```
sage: Q = Quiver.from_string("a--b-3,a---3,3-a")
sage: Q.adjacency_matrix()
[0 2 3]
[0 0 1]
[1 0 0]
sage: Q.vertices()
[0, 1, 2]
```

The actual labeling we use doesn't matter for the isomorphism type of the quiver:

```
sage: from quiver import *
sage: Quiver.from_matrix([[0, 3], [0, 0]]) == Quiver.from_string("12---b")
True
```

However, it does influence the labels of the vertex if we choose so:

```
sage: Quiver.from_string("12---b", forget_labels=False).vertices()
[12, 'b']
sage: Quiver.from_string("foo---bar", forget_labels=False).vertices()
['foo', 'bar']
```

__str__() → str

Detailed description of the quiver

Everything you get from *Quiver.repr()* together with the adjacency matrix.

EXAMPLES:

The 3-Kronecker quiver:

```
sage: from quiver import *
sage: Q = Quiver.from_string("1---2"); print(Q)
a quiver with 2 vertices and 3 arrows
adjacency matrix:
[0 3]
[0 0]
sage: Q.rename("3-Kronecker quiver"); print(Q)
3-Kronecker quiver
adjacency matrix:
[0 3]
[0 0]
```

repr() → str

Basic description of the quiver

To override the output, one uses `Quiver.rename()` from the *Element* class. The output of *Quiver.repr()* is that of `Quiver.get_custom_name()` if it is set, else it is the default specifying the number of vertices and arrows.

OUTPUT: a basic description of the quiver

EXAMPLES:

The 3-Kronecker quiver:

```
sage: from quiver import *
sage: Q = Quiver.from_string("1---2"); Q
a quiver with 2 vertices and 3 arrows
```

(continues on next page)

(continued from previous page)

```
sage: Q.rename("3-Kronecker quiver"); Q
3-Kronecker quiver
```

Renaming and resetting the name:

```
sage: Q = Quiver.from_string("1---2")
sage: Q.get_custom_name() is None
True
sage: Q.rename("3-Kronecker quiver")
sage: Q.get_custom_name()
'3-Kronecker quiver'
sage: Q.reset_name()
sage: Q.get_custom_name() is None
True
sage: Q
a quiver with 2 vertices and 3 arrows
```

str() → str

Full description of the quiver

This combines the output of *Quiver.repr()* with the adjacency matrix.

OUTPUT: a complete description of the quiver

EXAMPLES:

The 3-Kronecker quiver:

```
sage: from quiver import *
sage: Q = Quiver.from_string("1---2"); print(Q)
a quiver with 2 vertices and 3 arrows
adjacency matrix:
[0 3]
[0 0]
sage: Q.rename("3-Kronecker quiver"); print(Q)
3-Kronecker quiver
adjacency matrix:
[0 3]
[0 0]
```

__eq__(*other*) → bool

Checks for equality of quivers.

Equality here refers to equality of adjacency matrices, but disregarding the name of the quiver.

INPUT:

- `other` – Quiver; the quiver to compare against

OUTPUT: whether the adjacency matrices are the same

EXAMPLES:

The 2-Kronecker quiver and the generalized Kronecker quiver are the same:

```
sage: from quiver import *
sage: KroneckerQuiver() == GeneralizedKroneckerQuiver(2)
True
```

adjacency_matrix()

Returns the adjacency matrix of the quiver.

OUTPUT: The square matrix `M` whose entry `M[i,j]` is the number of arrows from the vertex `i` to the vertex `j`

EXAMPLES:

The adjacency matrix of a quiver construct from an adjacency matrix:

```
sage: from quiver import *
sage: M = matrix([[0, 3], [0, 0]])
sage: M == Quiver(M).adjacency_matrix()
True
```

graph()

Return the underlying graph of the quiver

OUTPUT: the underlying quiver as a DiGraph object

EXAMPLES:

The underlying graph of the quiver from a directed graph is that graph:

```
sage: from quiver import *
sage: G = DiGraph(matrix([[0, 3], [0, 0]]))
sage: G == Quiver.from_digraph(G).graph()
True
```

vertices()

Return the vertices of the quiver

If the quiver is created from a DiGraph or string, the vertices are labelled using the data in the DiGraph or string, as explained in *Quiver.from_digraph()* or *Quiver.from_string()*. If the quiver is created from a matrix, the vertices are labelled from *0* to *n-1*, where *n* is the number of rows or columns in the matrix.

OUTPUT: the vertices in the underlying graph

EXAMPLES:

Usually the vertices will be just integers:

```
sage: from quiver import *
sage: Quiver([[0, 3], [0, 0]]).vertices()
[0, 1]
```

We can have non-trivial labels for a quiver:

```
sage: Quiver.from_string("foo---bar", forget_labels=False).vertices()
['foo', 'bar']
```

number_of_vertices() → int

Returns the number of vertices

OUTPUT: the number of vertices

EXAMPLES:

There are 3 vertices in a 3-vertex quiver:

```
sage: from quiver import *
sage: ThreeVertexQuiver(1, 2, 4).number_of_vertices()
3
```

number_of_arrows() → int

Returns the number of arrows

OUTPUT: the number of arrows

EXAMPLES:

There are 7 arrows in this 3-vertex quiver:

```
sage: from quiver import *
sage: ThreeVertexQuiver(1, 2, 4).number_of_arrows()
7
```

is_acyclic() → bool

Returns whether the quiver is acyclic.

OUTPUT: True if the quiver is acyclic, False otherwise.

EXAMPLES:

An acyclic graph:

```
sage: from quiver import *
sage: KroneckerQuiver(3).is_acyclic()
True
```

A non-acyclic graph:

```
sage: GeneralizedJordanQuiver(5).is_acyclic()
False
```

is_connected() → bool

Returns whether the underlying graph of the quiver is connected or not.

OUTPUT: True if the quiver is connected, False otherwise.

EXAMPLES:

The n-Kronecker quivers are connected:

```
sage: from quiver import *
sage: KroneckerQuiver(4).is_connected()
True
```

The loop quivers are connected:

```
sage: GeneralizedJordanQuiver(3).is_connected()
True
```

is_finite_type() → bool

Returns whether the quiver is of finite type representation type.

This is the case if and only the connected components of the underlying undirected graph are isomorphic to Dynkin diagrams.

EXAMPLES:

The generalized Kronecker quiver is finite only for $m = 1$:

```
sage: from quiver import *
sage: GeneralizedKroneckerQuiver(1).is_finite_type()
True
sage: GeneralizedKroneckerQuiver(2).is_finite_type()
False
sage: GeneralizedKroneckerQuiver(3).is_finite_type()
False
```

is_tame_type() → bool

Returns whether the quiver is of tame type representation type.

This is the case if and only the connected components of the underlying undirected graph are isomorphic to (extended) Dynkin diagrams, with at least one being extended Dynkin.

EXAMPLES:

The generalized Kronecker quiver is tame only for $m = 2$:

```
sage: from quiver import *
sage: GeneralizedKroneckerQuiver(1).is_tame_type()
False
sage: GeneralizedKroneckerQuiver(2).is_tame_type()
True
sage: GeneralizedKroneckerQuiver(3).is_tame_type()
False
```

is_wild_type() → bool

Returns whether the quiver is of wild type representation type.

This is the case if and only the connected components of the underlying undirected graph are not all isomorphic to (extended) Dynkin diagrams.

EXAMPLES:

The generalized Kronecker quiver is wild for all $m \geq 3$:

```
sage: from quiver import *
sage: GeneralizedKroneckerQuiver(1).is_wild_type()
False
sage: GeneralizedKroneckerQuiver(2).is_wild_type()
False
sage: GeneralizedKroneckerQuiver(3).is_wild_type()
True
```

in_degree(*i*)

Returns the in-degree of a vertex.

The in-degree of `i` is the number of incoming arrows at `i`.

The parameter `i` must be an element of the vertices of the underlying graph. If constructed from a matrix or string, `i` can go from *0* to *n-1* where *n* is the number of vertices in the graph.

INPUT:

- `i` – a vertex of the underlying graph

OUTPUT: The in-degree of the vertex `i`

EXAMPLES:

In the 3-Kronecker quiver the in-degree is either 0 or 3:

```
sage: from quiver import *
sage: Q = GeneralizedKroneckerQuiver(3)
sage: Q.in_degree(0)
0
sage: Q.in_degree(1)
3
```

If we specified a non-standard labeling on the vertices we must use it:

```
sage: Q = Quiver.from_string("a---b", forget_labels=False)
sage: Q.in_degree("a")
0
sage: Q.in_degree("b")
3
```

out_degree(*i*)

Returns the out-degree of a vertex.

The parameter `i` must be an element of the vertices of the underlying graph. If constructed from a matrix or string, `i` can go from *0* to *n-1* where *n* is the number of vertices in the graph.

The out-degree of `i` is the number of outgoing arrows at `i`.

INPUT:

- `i` – a vertex of the underlying graph

OUTPUT: The out-degree of the vertex `i`

EXAMPLES:

In the 3-Kronecker quiver the out-degree is either 3 or 0:

```
sage: from quiver import *
sage: Q = GeneralizedKroneckerQuiver(3)
sage: Q.out_degree(0)
3
sage: Q.out_degree(1)
0
```

If we specified a non-standard labeling on the vertices we must use it:

```
sage: Q = Quiver.from_string("a---b", forget_labels=False)
sage: Q.out_degree("a")
3
sage: Q.out_degree("b")
0
```

is_source(*i*) → bool

Checks if `i` is a source of the quiver

The vertex `i` is a source if there are no incoming arrows at `i`.

INPUT:

- `i` – a vertex of the quiver

OUTPUT: whether `i` is a source of the quiver

EXAMPLES:

The 3-Kronecker quiver has one source:

```
sage: from quiver import *
sage: Q = GeneralizedKroneckerQuiver(3)
sage: Q.is_source(0)
True
sage: Q.is_source(1)
False
```

If we specified a non-standard labeling on the vertices we must use it:

```
sage: Q = Quiver.from_string("a---b", forget_labels=False)
sage: Q.is_source("a")
True
sage: Q.is_source("b")
False
```

is_sink(i) $\to$ bool

Checks if `i` is a sink of the quiver

The vertex `i` is a sink if there are no outgoing arrows out of `i`.

INPUT:

- `i` – a vertex of the quiver

OUTPUT: whether `i` is a sink of the quiver

EXAMPLES

The 3-Kronecker quiver has one sink:

```
sage: from quiver import *
sage: Q = GeneralizedKroneckerQuiver(3)
sage: Q.is_sink(0)
False
sage: Q.is_sink(1)
True
```

If we specified a non-standard labeling on the vertices we must use it:

```
sage: Q = Quiver.from_string("a---b", forget_labels=False)
sage: Q.is_sink("a")
False
sage: Q.is_sink("b")
True
```

sources()

Return the vertices which are sources in the quiver

OUTPUT: the list of vertices without incoming edges

EXAMPLES:

The 3-Kronecker quiver has one source:

```
sage: from quiver import *
sage: GeneralizedKroneckerQuiver(3).sources()
[0]
```

It is possible that a quiver has no sources:

```
sage: JordanQuiver().sources()
[]
```

sinks()

Return the vertices which are sinks in the quiver

OUTPUT: the list of vertices without incoming edges

EXAMPLES:

The 3-Kronecker quiver has one source:

```
sage: from quiver import *
sage: GeneralizedKroneckerQuiver(3).sources()
[0]
```

It is possible that a quiver has no sinks:

```
sage: JordanQuiver().sinks()
[]
```

euler_matrix()

Returns the Euler matrix of the quiver

This is the matrix representing the Euler form, defined by

$$\langle \mathbf{d}, \mathbf{e} \rangle = \sum_{i \in Q_0} d_i e_i - \sum_{\alpha \in Q_1} d_{s(\alpha)} e_{t(\alpha)}$$

In the basis given by the vertices, it can be written as the difference of the identity matrix and the adjacency matrix.

OUTPUT: the Euler matrix of the quiver

EXAMPLES:

The Kronecker 3-quiver:

```
sage: from quiver import *
sage: GeneralizedKroneckerQuiver(3).euler_matrix()
[ 1 -3]
[ 0  1]
```

It uses the basis of the vertices, so it agrees with this alternative definition:

```
sage: Quiver.from_string("foo---bar", forget_labels=False).euler_matrix()
[ 1 -3]
[ 0  1]
```

euler_form(x, y) $\to$ int

The value $\langle x, y \rangle$ of the Euler form

INPUT:

- `x` – an element of $\mathbb{Z}Q_0$
- `y` – an element of $\mathbb{Z}Q_0$

OUTPUT: the value of the Euler form, i.e., `x * self.euler_matrix() * y`

EXAMPLES:

An example using the Kronecker quiver:

```
sage: from quiver import *
sage: Q = GeneralizedKroneckerQuiver(3)
sage: Q.euler_form((1, 3), (2, -2))
2
```

It uses the basis of the vertices, so we specify the entries of elements of $\mathbb{Z}Q_0$ in this order, thus the same example as before:

```
sage: Q = Quiver.from_string("foo---bar", forget_labels=False)
sage: Q.euler_form((1, 3), (2, -2))
2
```

cartan_matrix()

Returns the Cartan matrix of the quiver

This is the matrix representing the symmetrization of the Euler form, see *Quiver.euler_matrix()*

OUTPUT: the Cartan matrix of the quiver

EXAMPLES:

The Kronecker 3-quiver:

```
sage: from quiver import *
sage: GeneralizedKroneckerQuiver(3).cartan_matrix()
[ 2 -3]
[-3  2]
```

symmetrized_euler_form(x, y) → int

The value (x, y) of the Euler form

INPUT:

- x – an element of $\mathbb{Z}Q_0$
- y – an element of $\mathbb{Z}Q_0$

OUTPUT: the value of the symmetrized Euler form applied to x and y

EXAMPLES:

An example using the Kronecker quiver:

```
sage: from quiver import *
sage: Q = GeneralizedKroneckerQuiver(3)
sage: Q.symmetrized_euler_form((1, 3), (2, -2))
-20
```

It uses the basis of the vertices, so we specify the entries of elements of $\mathbb{Z}Q_0$ in this order, thus the same example as before:

```
sage: Q = Quiver.from_string("foo---bar", forget_labels=False)
sage: Q.symmetrized_euler_form((1, 3), (2, -2))
-20
```

tits_form(x) → int

The value of the Tits quadratic form of the quiver at x

This is really just the value $\langle x, x\rangle$ of the Euler form, or half of the value (x, x) of the symmetrized Euler form.

INPUT:

- x – an element of $\mathbb{Z}Q_0$

OUTPUT: the value of the Tits form applied to x

EXAMPLES:

An example using the Kronecker quiver:

```
sage: from quiver import *
sage: Q = GeneralizedKroneckerQuiver(3)
sage: Q.tits_form((2, 3))
-5
```

It uses the basis of the vertices, so we specify the entries of elements of $\mathbb{Z}Q_0$ in this order, thus the same example as before:

```
sage: Q = Quiver.from_string("foo---bar", forget_labels=False)
sage: Q.tits_form((2, 3))
-5
```

opposite_quiver()

Returns the opposite quiver

The opposite quiver is the quiver with all arrows reversed. Its adjacency matrix is given by the transpose of the adjacency matrix.

OUTPUT: the opposite quiver

EXAMPLES:

The opposite of the 3-Kronecker quiver:

```
sage: from quiver import *
sage: print(GeneralizedKroneckerQuiver(3).opposite_quiver())
opposite of 3-Kronecker quiver
adjacency matrix:
[0 0]
[3 0]
```

It preserves the labelling of the vertices:

```
sage: Q = Quiver.from_string("foo---bar", forget_labels=False)
sage: Qopp = Q.opposite_quiver()
sage: Qopp.vertices()
['foo', 'bar']
sage: Qopp.adjacency_matrix()
[0 0]
[3 0]
```

doubled_quiver()

Returns the doubled quiver

The double of a quiver is the quiver where for each arrow we add an arrow in the opposite direction.

Its adjacency matrix is the sum of the adjacency matrix of the original quiver and its transpose.

OUTPUT: the doubled quiver

EXAMPLES:

The double of the 3-Kronecker quiver:

```
sage: from quiver import *
sage: print(GeneralizedKroneckerQuiver(3).doubled_quiver())
double of 3-Kronecker quiver
adjacency matrix:
[0 3]
[3 0]
```

It preserves the labelling of the vertices:

```
sage: Q = Quiver.from_string("foo---bar", forget_labels=False)
sage: Qbar = Q.doubled_quiver()
sage: Qbar.vertices()
['foo', 'bar']
sage: Qbar.adjacency_matrix()
[0 3]
[3 0]
```

framed_quiver (*framing*, *vertex='-oo'*)

Returns the framed quiver with framing vector `framing`

The optional parameter `vertex` determines the name of the framing vertex, which defaults to *-oo*.

The framed quiver has one additional vertex, and *f_i* many arrows from the framing vertex to *i*, for every *iin Q_0*.

INPUT:

- `framing` – list of non-negative integers saying how many arrows from the framed vertex to `i`
- `vertex` (default: "-oo") – name of the framing vertex

OUTPUT: the framed quiver

EXAMPLES:

Framing the 3-Kronecker quiver:

```
sage: from quiver import *
sage: Q = GeneralizedKroneckerQuiver(3).framed_quiver([1, 0])
sage: print(Q)
framing of 3-Kronecker quiver
adjacency matrix:
[0 1 0]
[0 0 3]
[0 0 0]
sage: Q.vertices()
['-oo', 0, 1]
sage: Q = GeneralizedKroneckerQuiver(3).framed_quiver([2, 2], vertex="a")
sage: print(Q)
framing of 3-Kronecker quiver
adjacency matrix:
[0 2 2]
[0 0 3]
[0 0 0]
sage: Q.vertices()
['a', 0, 1]
```

If you frame twice it will have to use a different vertex label:

```
sage: Q = GeneralizedKroneckerQuiver(3).framed_quiver([2, 2])
sage: Q.framed_quiver([1, 1, 1]).vertices()
Traceback (most recent call last):
...
ValueError: -oo is already a vertex
```

coframed_quiver(*coframing*, *vertex='+oo'*)

Returns the coframed quiver with coframing vector `coframing`

The optional parameter `vertex` determines the name of the coframing vertex, which defaults to *+oo*.

The coframed quiver has one additional vertex, and *f_i* many arrows from the vertex *i* to the coframed vertex, for every $i \in Q_0$.

INPUT:

- `coframing` – list of non-negative integers saying how many arrows go from the framed vertex to *i*
- `vertex` (default: None) – name of the framing vertex

OUTPUT: the framed quiver

EXAMPLES:

Coframing the 3-Kronecker quiver:

```
sage: from quiver import *
sage: Q = GeneralizedKroneckerQuiver(3).coframed_quiver([1, 0])
sage: print(Q)
coframing of 3-Kronecker quiver
adjacency matrix:
[0 3 1]
[0 0 0]
[0 0 0]
sage: Q.vertices()
[0, 1, '+oo']
sage: Q = GeneralizedKroneckerQuiver(3).coframed_quiver([2, 2], vertex="a")
sage: print(Q)
coframing of 3-Kronecker quiver
adjacency matrix:
[0 3 2]
[0 0 2]
[0 0 0]
sage: Q.vertices()
[0, 1, 'a']
```

If you coframe twice it will have to use a different vertex label:

```
sage: Q = GeneralizedKroneckerQuiver(3).coframed_quiver([2, 2])
sage: Q.coframed_quiver([1, 1, 1]).vertices()
Traceback (most recent call last):
...
ValueError: +oo is already a vertex
```

full_subquiver(*vertices*)

Returns the full subquiver supported on the given set of vertices

INPUT:

- `vertices`: list of vertices for the subquiver

OUTPUT: the full subquiver on the specified vertices

EXAMPLES:

Some basic examples:

```
sage: from quiver import *
sage: Q = ThreeVertexQuiver(2, 3, 4)
sage: print(Q.full_subquiver([0, 1]))
full subquiver of an acyclic 3-vertex quiver of type (2, 3, 4)
adjacency matrix:
[0 2]
[0 0]
sage: print(Q.full_subquiver([0, 2]))
full subquiver of an acyclic 3-vertex quiver of type (2, 3, 4)
adjacency matrix:
[0 3]
[0 0]
```

If we specified a non-standard labeling on the vertices we must use it:

```
sage: Q = Quiver.from_string("a--b----c,a---c", forget_labels=False)
sage: Q == ThreeVertexQuiver(2, 3, 4)
True
sage: print(Q.full_subquiver(["a", "b"]))
a quiver with 2 vertices and 2 arrows
adjacency matrix:
[0 2]
[0 0]
sage: print(Q.full_subquiver(["a", "c"]))
a quiver with 2 vertices and 3 arrows
adjacency matrix:
[0 3]
[0 0]
```

zero_vector()

Returns the zero dimension vector.

The output is adapted to the vertices.

OUTPUT: the zero dimension vector

EXAMPLES:

Usually it is an actual vector:

```
sage: from quiver import *
sage: KroneckerQuiver(3).zero_vector()
(0, 0)
sage: type(KroneckerQuiver(3).zero_vector())
<class 'sage.modules.vector_integer_dense.Vector_integer_dense'>
```

But if the quiver has custom vertex labels it is a dict:

```
sage: Q = Quiver.from_string("a--b----c,a---c", forget_labels=False)
sage: Q.zero_vector()
{'a': 0, 'b': 0, 'c': 0}
```

thin_dimension_vector()

Returns the thin dimension vector, i.e., all ones

The output is adapted to the vertices.

OUTPUT: the thin dimension vector

EXAMPLES:

Usually it is an actual vector:

```
sage: from quiver import *
sage: KroneckerQuiver(3).thin_dimension_vector()
(1, 1)
sage: type(KroneckerQuiver(3).thin_dimension_vector())
<class 'sage.modules.vector_integer_dense.Vector_integer_dense'>
```

But if the quiver has custom vertex labels it is a dict:

```
sage: Q = Quiver.from_string("a--b----c,a---c", forget_labels=False)
sage: Q.thin_dimension_vector()
{'a': 1, 'b': 1, 'c': 1}
```

simple_root (*i*)

Returns the simple root at the vertex `i`

The output is adapted to the vertices.

OUTPUT: the simple root at the vertex `i`

EXAMPLES:

Usually it is an actual vector:

```
sage: from quiver import *
sage: KroneckerQuiver(3).simple_root(1)
(0, 1)
sage: type(KroneckerQuiver(3).simple_root(1))
<class 'sage.modules.vector_integer_dense.Vector_integer_dense'>
```

But if the quiver has custom vertex labels it is a dict:

```
sage: Q = Quiver.from_string("a--b----c,a---c", forget_labels=False)
sage: Q.simple_root("b")
{'a': 0, 'b': 1, 'c': 0}
```

is_root (*x*) → bool

Checks whether `x` is a root of the underlying diagram of the quiver.

A root is a non-zero vector x in $\mathbb{Z}Q_0$ such that the Tits form of x is at most 1.

INPUT:

- `x`: integer vector

OUTPUT: whether `x` is a root

EXAMPLES:

Some roots and non-roots for the 3-Kronecker quiver:

```
sage: from quiver import *
sage: Q = KroneckerQuiver(3)
sage: Q.is_root((2, 3))
True
```

(continues on next page)

(continued from previous page)

```
sage: Q.is_root(Q.zero_vector())
False
sage: Q.is_root((4, 1))
False
```

is_real_root(x) $\to$ bool

Checks whether x is a real root of the underlying diagram of the quiver.

A root is called real if its Tits form equals 1.

INPUT:

- x: integer vector

OUTPUT: whether x is a real root

EXAMPLES:

Some real and non-real for the 3-Kronecker quiver:

```
sage: from quiver import *
sage: Q = KroneckerQuiver(3)
sage: Q.is_real_root((2, 3))
False
sage: Q.is_real_root(Q.zero_vector())
False
sage: Q.is_real_root((3, 1))
True
```

is_imaginary_root(x) $\to$ bool

Checks whether x is a imaginary root of the quiver.

A root is called imaginary if its Tits form is non-positive.

INPUT:

- x: integer vector

OUTPUT: whether x is an imaginary root

EXAMPLES:

Some imaginary roots and non imaginary roots for the 3-Kronecker quiver:

```
sage: from quiver import *
sage: Q = KroneckerQuiver(3)
sage: Q.is_imaginary_root((2, 3))
True
sage: Q.is_imaginary_root(Q.zero_vector())
False
sage: Q.is_imaginary_root((4, 1))
False
```

is_schur_root(d) $\to$ bool

Checks if d is a Schur root.

INPUT:

- d: dimension vector

OUTPUT: whether `d` is an imaginary root

A Schur root is a dimension vector which admits a Schurian representation, i.e., a representation whose endomorphism ring is the field itself. It is necessarily indecomposable.

By MR1162487 **d** is a Schur root if and only if it admits a stable representation for the canonical stability parameter.

EXAMPLES:

The dimension vector *(2, 3)* is Schurian for the 3-Kronecker quiver:

```
sage: from quiver import *
sage: Q = GeneralizedKroneckerQuiver(3)
sage: Q.is_schur_root([2, 3])
True
```

Examples from Derksen–Weyman's book (Example 11.1.4):

```
sage: from quiver import *
sage: Q = ThreeVertexQuiver(1, 1, 1)
sage: Q.is_schur_root((1, 1, 2))
True
sage: Q.is_schur_root((1, 2, 1))
False
sage: Q.is_schur_root((1, 1, 1))
True
sage: Q.is_schur_root((2, 2, 2))
False
```

slope (*d*, *theta=None*, *denom=<built-in function sum>*)

Returns the slope of `d` with respect to `theta`

The slope is defined as the value of *theta(d)* divided by the total dimension of *d*. It is possible to vary the denominator, to use a function more general than the sum.

INPUT:

- `d` – dimension vector
- `theta` – (default: canonical stability parameter) stability parameter
- `denom` – (default: sum) the denominator function

OUTPUT: the slope of `d` with respect to `theta` and optional `denom`

EXAMPLES:

Some slopes for the Kronecker quiver, first for the canonical stability parameter, then for some other:

```
sage: from quiver import *
sage: Q = KroneckerQuiver(3)
sage: d = (2, 3)
sage: Q.slope(d, (9, -6))
0
sage: Q.slope(d)
0
sage: Q.slope(d, (2, -2))
-2/5
```

We can use for instance a constant denominator:

```
sage: constant = lambda di: 1
sage: Q.slope(d, Q.canonical_stability_parameter(d), denom=constant)
0
```

The only dependence on the quiver is the set of vertices, so if we don't use vertex labels, the choice of quiver doesn't matter:

```
sage: d, theta = (2, 3), (9, -6)
sage: KroneckerQuiver(3).slope(d, theta)
0
```

is_subdimension_vector(*e*, *d*)

Determine whether `e` is a subdimension vector of `d`

INPUT:

– `e` – dimension vector

– `d` – dimension vector

OUTPUT: whether `e` is a subdimension vector of `d`

EXAMPLES:

Some basic examples:

```
sage: from quiver import *
sage: Q = KroneckerQuiver(3)
sage: Q.is_subdimension_vector((1, 2), (2, 3))
True
sage: Q.is_subdimension_vector((2, 3), (2, 3))
True
sage: Q.is_subdimension_vector((6, 6), (2, 3))
False
```

We can also work with vertex labels:

```
sage: Q = Quiver.from_string("a--b----c,a---c", forget_labels=False)
sage: d = {"a" : 3, "b" : 3, "c" : 3}
sage: e = {"a" : 1, "b" : 2, "c" : 3}
sage: Q.is_subdimension_vector(e, d)
True
sage: Q.is_subdimension_vector(d, e)
False
```

all_subdimension_vectors(*d*, *proper=False*, *nonzero=False*, *forget_labels=False*)

Returns the list of all subdimension vectors of d.

INPUT:

- `d` – dimension vector
- `proper` (default: False) – whether to exclude `d`
- `nonzero` (default: False) – whether to exclude the zero vector
- `forget_labels` (default: False) – whether to forget the vertex labels

OUTPUT: all subdimension vectors of `d` (maybe excluding zero and/or `d`)

EXAMPLES:

The usual use cases:

```
sage: from quiver import *
sage: Q = KroneckerQuiver(3)
sage: Q.all_subdimension_vectors((2, 3))
    [(0, 0),
     (0, 1),
     (0, 2),
     (0, 3),
     (1, 0),
     (1, 1),
     (1, 2),
     (1, 3),
     (2, 0),
     (2, 1),
     (2, 2),
     (2, 3)]
sage: Q.all_subdimension_vectors((2, 3), proper=True)
    [(0, 0),
     (0, 1),
     (0, 2),
     (0, 3),
     (1, 0),
     (1, 1),
     (1, 2),
     (1, 3),
     (2, 0),
     (2, 1),
     (2, 2)]
sage: Q.all_subdimension_vectors((2, 3), nonzero=True)
    [(0, 1),
     (0, 2),
     (0, 3),
     (1, 0),
     (1, 1),
     (1, 2),
     (1, 3),
     (2, 0),
     (2, 1),
     (2, 2),
     (2, 3)]
sage: Q.all_subdimension_vectors((2, 3), proper=True, nonzero=True)
    [(0, 1),
     (0, 2),
     (0, 3),
     (1, 0),
     (1, 1),
     (1, 2),
     (1, 3),
     (2, 0),
     (2, 1),
     (2, 2)]
```

Some exceptional cases:

```
sage: Q.all_subdimension_vectors(Q.zero_vector())
[(0, 0)]
sage: Q.all_subdimension_vectors(Q.zero_vector(), proper=True)
```

(continues on next page)

(continued from previous page)

```
[]
```

If we work with labeled vertices, then we get a list of dicts:

```
sage: Q = Quiver.from_string("a---b", forget_labels=False)
sage: Q.all_subdimension_vectors((1, 2))
[{'a': 0, 'b': 0},
 {'a': 0, 'b': 1},
 {'a': 0, 'b': 2},
 {'a': 1, 'b': 0},
 {'a': 1, 'b': 1},
 {'a': 1, 'b': 2}]
sage: Q.all_subdimension_vectors((1, 2), forget_labels=True)
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2)]
```

is_theta_coprime(*d*, *theta=None*) → bool

Checks if `d` is `theta`-coprime.

A dimension vector d is θ-coprime if $\mu_\theta(e) \neq \mu_\theta(e)$ for all proper non-zero subdimension vectors e of d.

The default value for `theta` is the canonical stability parameter.

INPUT:

- `d` – dimension vector
- `theta` – (default: canonical stability paramter) stability parameter

EXAMPLES:

Examples of coprimality:

```
sage: from quiver import *
sage: Q = KroneckerQuiver(3)
sage: d = (2, 3)
sage: Q.is_theta_coprime(d, Q.canonical_stability_parameter(d))
True
sage: Q.is_theta_coprime(d)
True
sage: Q.is_theta_coprime((3, 3), (1, -1))
False
```

is_indivisible(*d*) → bool

Checks if the gcd of all entries of `d` is 1

INPUT:

– `d` – dimension vector

OUTPUT: whether the dimension vector is indivisible

EXAMPLES:

Two examples with the Kronecker quiver:

```
sage: from quiver import *
sage: Q = KroneckerQuiver(3)
sage: Q.is_indivisible((2, 3))
True
sage: Q.is_indivisible((2, 2))
False
```

support(d)

Returns the support of the dimension vector.

INPUT:

- d: dimension vector

OUTPUT: subset of vertices in the underlying graph in the support

The support is the set $\{i \in Q_0 \mid d_i > 0\}$.

EXAMPLES:

The support is the set of vertices for which the value of the dimension vector is nonzero:

```
sage: from quiver import *
sage: Q = ThreeVertexQuiver(2, 0, 4)
sage: d = (1, 1, 1)
sage: Q.support(d)
[0, 1, 2]
sage: d = (1, 0, 1)
sage: Q.support(d)
[0, 2]
```

It takes into account vertex labels:

```
sage: Q = Quiver.from_string("a--b----c,a---c", forget_labels=False)
sage: d = {"a": 2, "b": 3, "c": 0}
sage: Q.support(d)
['a', 'b']
```

in_fundamental_domain(d, *depth=0*)

Checks if a dimension vector is in the fundamental domain.

The fundamental domain of Q is the set of dimension vectors d such that

- supp($\mathbf{d}$) is connected
- $\langle d, e_i\rangle + \langle e_i, d\rangle \leq 0$ for every simple root

Every d in the fundamental domain is an imaginary root and the set of imaginary roots is the Weyl group saturation of the fundamental domain. If d is in the fundamental domain then it is Schurian and a general representation of dimension vector d is stable for the canonical stability parameter.

The optional parameter depth allows to make the inequality stricter.

INPUT:

- d: dimension vector
- depth (default: 0) – how deep the vector should be in the domain

OUTPUT: whether d is in the (interior of) the fundamental domain

EXAMPLES:

The fundamental domain of the 3-Kronecker quiver:

```
sage: from quiver import *
sage: Q = GeneralizedKroneckerQuiver(3)
sage: Q.in_fundamental_domain((1, 1))
True
sage: Q.in_fundamental_domain((1, 2))
```

(continues on next page)

(continued from previous page)

```
False
sage: Q.in_fundamental_domain((2, 3))
True
```

The same calculation now with vertex labels:

```
sage: Q = Quiver.from_string("a---b", forget_labels=False)
sage: Q.in_fundamental_domain({"a" : 1, "b" : 1})
True
sage: Q.in_fundamental_domain({"a" : 1, "b" : 2})
False
sage: Q.in_fundamental_domain({"a" : 2, "b" : 3})
True
```

We test for dimension vectors in the strict interior, where the depth is equal to 1:

```
sage: from quiver import *
sage: Q = GeneralizedKroneckerQuiver(3)
sage: Q.in_fundamental_domain((1, 1), depth=1)
True
sage: Q.in_fundamental_domain((2, 3), depth=1)
False
```

division_order(d, e)

Checks if $d \ll e$

This means that

- $d_i \leq e_i$ for every source i
- $d_j \geq e_j$ for every sink j,
- $d_k = e_k$ for every vertex k which is neither a source nor a sink.

This is used when dealing with Chow rings of quiver moduli, see also *QuiverModuli.chow_ring()* and `QuiverModuli._all_minimal_forbidden_subdimension_vectors()`.

EXAMPLES:

The division order on some dimension vectors for the 3-Kronecker quiver:

```
sage: from quiver import *
sage: Q = GeneralizedKroneckerQuiver(3)
sage: d = (1, 1)
sage: e = (2, 1)
sage: f = (2, 2)
sage: Q.division_order(d, e)
True
sage: Q.division_order(e, d)
False
sage: Q.division_order(d, f)
False
sage: Q.division_order(f, d)
False
sage: Q.division_order(e, f)
False
sage: Q.division_order(f, e)
True
```

The division order on some dimension vectors for a 3-vertex quiver:

```
sage: Q = ThreeVertexQuiver(2, 2, 2)
sage: d = (1, 1, 1)
sage: e = (1, 2, 1)
sage: Q.division_order(d, e)
False
sage: Q.division_order(e, d)
False
```

is_generic_subdimension_vector$(e, d) \to$ bool

Checks if e is a generic subdimension vector of d.

INPUT:

- e: dimension vector for the subrepresentation
- d: dimension vector for the ambient representation

OUTPUT: whether e is a generic subdimension vector of d

A dimension vector e is a generic subdimension vector of d if a generic representation of dimension vector d possesses a subrepresentation of dimension vector e. By MR1162487 e is a generic subdimension vector of d if and only if e is a subdimension vector of d and $\langle f, d - e\rangle$ is non-negative for all generic subdimension vectors f of e.

EXAMPLES:

Some examples on loop quivers:

```
sage: from quiver import *
sage: Q = LoopQuiver(1)
sage: ds = [vector([i]) for i in range(3)]
sage: for (e, d) in cartesian_product([ds, ds]):
....:     if not Q.is_subdimension_vector(e, d): continue
....:     print("{} is generic subdimension vector of {}: {}".format(
....:         e, d, Q.is_generic_subdimension_vector(e,d))
....:     )
(0) is generic subdimension vector of (0): True
(0) is generic subdimension vector of (1): True
(0) is generic subdimension vector of (2): True
(1) is generic subdimension vector of (1): True
(1) is generic subdimension vector of (2): True
(2) is generic subdimension vector of (2): True
sage: Q = LoopQuiver(2)
sage: for (e, d) in cartesian_product([ds]*2):
....:     if not Q.is_subdimension_vector(e, d): continue
....:     print("{} is generic subdimension vector of {}: {}".format(
....:         e, d, Q.is_generic_subdimension_vector(e,d))
....:     )
(0) is generic subdimension vector of (0): True
(0) is generic subdimension vector of (1): True
(0) is generic subdimension vector of (2): True
(1) is generic subdimension vector of (1): True
(1) is generic subdimension vector of (2): False
(2) is generic subdimension vector of (2): True
```

Some examples on generalized Kronecker quivers:

```
sage: Q = GeneralizedKroneckerQuiver(1)
sage: ds = Tuples(range(3), 2)
sage: for (e, d) in cartesian_product([ds]*2):
....:     if not Q.is_subdimension_vector(e, d): continue
....:     print("{} is generic subdimension vector of {}: {}".format(
....:         e, d, Q.is_generic_subdimension_vector(e,d))
....:     )
(0, 0) is generic subdimension vector of (0, 0): True
(0, 0) is generic subdimension vector of (1, 0): True
(0, 0) is generic subdimension vector of (2, 0): True
(0, 0) is generic subdimension vector of (0, 1): True
(0, 0) is generic subdimension vector of (1, 1): True
(0, 0) is generic subdimension vector of (2, 1): True
(0, 0) is generic subdimension vector of (0, 2): True
(0, 0) is generic subdimension vector of (1, 2): True
(0, 0) is generic subdimension vector of (2, 2): True
(1, 0) is generic subdimension vector of (1, 0): True
(1, 0) is generic subdimension vector of (2, 0): True
(1, 0) is generic subdimension vector of (1, 1): False
(1, 0) is generic subdimension vector of (2, 1): True
(1, 0) is generic subdimension vector of (1, 2): False
(1, 0) is generic subdimension vector of (2, 2): False
(2, 0) is generic subdimension vector of (2, 0): True
(2, 0) is generic subdimension vector of (2, 1): False
(2, 0) is generic subdimension vector of (2, 2): False
(0, 1) is generic subdimension vector of (0, 1): True
(0, 1) is generic subdimension vector of (1, 1): True
(0, 1) is generic subdimension vector of (2, 1): True
(0, 1) is generic subdimension vector of (0, 2): True
(0, 1) is generic subdimension vector of (1, 2): True
(0, 1) is generic subdimension vector of (2, 2): True
(1, 1) is generic subdimension vector of (1, 1): True
(1, 1) is generic subdimension vector of (2, 1): True
(1, 1) is generic subdimension vector of (1, 2): True
(1, 1) is generic subdimension vector of (2, 2): True
(2, 1) is generic subdimension vector of (2, 1): True
(2, 1) is generic subdimension vector of (2, 2): False
(0, 2) is generic subdimension vector of (0, 2): True
(0, 2) is generic subdimension vector of (1, 2): True
(0, 2) is generic subdimension vector of (2, 2): True
(1, 2) is generic subdimension vector of (1, 2): True
(1, 2) is generic subdimension vector of (2, 2): True
(2, 2) is generic subdimension vector of (2, 2): True
sage: Q = GeneralizedKroneckerQuiver(2)
sage: for (e, d) in cartesian_product([ds]*2):
....:     if not Q.is_subdimension_vector(e, d): continue
....:     print("{} is generic subdimension vector of {}: {}".format(
....:         e, d, Q.is_generic_subdimension_vector(e,d))
....:     )
(0, 0) is generic subdimension vector of (0, 0): True
(0, 0) is generic subdimension vector of (1, 0): True
(0, 0) is generic subdimension vector of (2, 0): True
(0, 0) is generic subdimension vector of (0, 1): True
(0, 0) is generic subdimension vector of (1, 1): True
(0, 0) is generic subdimension vector of (2, 1): True
(0, 0) is generic subdimension vector of (0, 2): True
(0, 0) is generic subdimension vector of (1, 2): True
```

(continues on next page)

(continued from previous page)

```
(0, 0) is generic subdimension vector of (2, 2): True
(1, 0) is generic subdimension vector of (1, 0): True
(1, 0) is generic subdimension vector of (2, 0): True
(1, 0) is generic subdimension vector of (1, 1): False
(1, 0) is generic subdimension vector of (2, 1): False
(1, 0) is generic subdimension vector of (1, 2): False
(1, 0) is generic subdimension vector of (2, 2): False
(2, 0) is generic subdimension vector of (2, 0): True
(2, 0) is generic subdimension vector of (2, 1): False
(2, 0) is generic subdimension vector of (2, 2): False
(0, 1) is generic subdimension vector of (0, 1): True
(0, 1) is generic subdimension vector of (1, 1): True
(0, 1) is generic subdimension vector of (2, 1): True
(0, 1) is generic subdimension vector of (0, 2): True
(0, 1) is generic subdimension vector of (1, 2): True
(0, 1) is generic subdimension vector of (2, 2): True
(1, 1) is generic subdimension vector of (1, 1): True
(1, 1) is generic subdimension vector of (2, 1): True
(1, 1) is generic subdimension vector of (1, 2): False
(1, 1) is generic subdimension vector of (2, 2): True
(2, 1) is generic subdimension vector of (2, 1): True
(2, 1) is generic subdimension vector of (2, 2): False
(0, 2) is generic subdimension vector of (0, 2): True
(0, 2) is generic subdimension vector of (1, 2): True
(0, 2) is generic subdimension vector of (2, 2): True
(1, 2) is generic subdimension vector of (1, 2): True
(1, 2) is generic subdimension vector of (2, 2): True
(2, 2) is generic subdimension vector of (2, 2): True
```

all_generic_subdimension_vectors(*d*, *proper=False*, *nonzero=False*)

Returns the list of all generic subdimension vectors of `d`.

INPUT:

- `d`: dimension vector

OUTPUT: list of vectors

EXAMPLES:

Some n-Kronecker quivers:

```
sage: from quiver import *
sage: Q = GeneralizedKroneckerQuiver(1)
sage: d = (3, 3)
sage: Q.all_generic_subdimension_vectors(d)
[(0, 0),
 (0, 1),
 (0, 2),
 (0, 3),
 (1, 1),
 (1, 2),
 (1, 3),
 (2, 2),
 (2, 3),
 (3, 3)]
sage: Q = GeneralizedKroneckerQuiver(2)
sage: Q.all_generic_subdimension_vectors(d)
```

(continues on next page)

(continued from previous page)

```
[(0, 0),
 (0, 1),
 (0, 2),
 (0, 3),
 (1, 1),
 (1, 2),
 (1, 3),
 (2, 2),
 (2, 3),
 (3, 3)]
sage: Q = GeneralizedKroneckerQuiver(3)
sage: Q.all_generic_subdimension_vectors(d)
[(0, 0), (0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3), (3, 3)]
sage: Q.all_generic_subdimension_vectors(d, nonzero=True)
[(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3), (3, 3)]
sage: Q.all_generic_subdimension_vectors(d, proper=True)
[(0, 0), (0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)]
```

generic_ext(d, e)

Computes ext(d, e).

INPUT:

- d: dimension vector
- e: dimension vector

OUTPUT: dimension of the generic ext

According to Theorem 5.4 in Schofield's 'General representations of quivers', we have

$$\text{ext}(a, b) = \max\{-\langle c, b\rangle\}.$$

where c runs over the generic subdimension vectors of a.

EXAMPLES:

Generic ext on the 3-Kronecker quiver:

```
sage: from quiver import *
sage: Q = GeneralizedKroneckerQuiver(3)
sage: ds = [Q.simple_root(0), Q.simple_root(1), Q.thin_dimension_vector()]
sage: for (d, e) in cartesian_product([ds]*2):
....:     print("ext({}, {}) = {}".format(d, e, Q.generic_ext(d, e)))
ext((1, 0), (1, 0)) = 0
ext((1, 0), (0, 1)) = 3
ext((1, 0), (1, 1)) = 2
ext((0, 1), (1, 0)) = 0
ext((0, 1), (0, 1)) = 0
ext((0, 1), (1, 1)) = 0
ext((1, 1), (1, 0)) = 0
ext((1, 1), (0, 1)) = 2
ext((1, 1), (1, 1)) = 1
```

generic_hom(d, e)

Computes hom(d, e).

INPUT:

- d: dimension vector

- e: dimension vector

OUTPUT: dimension of the generic hom

There is a non-empty open subset U of $R(Q,d) \times R(Q,e)$ such that

$$\dim(\mathrm{Ext}(M,N)) = \mathrm{ext}(d,e),$$

i.e., $\dim(\mathrm{Ext}(M,N))$ is minimal for all *(M,N)* in *U*.

Therefore, *operatorname{dim}(operatorname{Hom}(M,N)) = langle a,brangle + operatorname{dim}(operatorname{Ext}(M,N))* is minimal, and *operatorname{hom}(a,b) = langle a,brangle + operatorname{ext}(a,b)*.

EXAMPLES:

Generic hom on the 3-Kronecker quiver:

```
sage: from quiver import *
sage: Q = GeneralizedKroneckerQuiver(3)
sage: ds = [Q.simple_root(0), Q.simple_root(1), Q.thin_dimension_vector()]
sage: for (d, e) in cartesian_product([ds]*2):
....:     print("hom({}, {}) = {}".format(d, e, Q.generic_hom(d, e)))
hom((1, 0), (1, 0)) = 1
hom((1, 0), (0, 1)) = 0
hom((1, 0), (1, 1)) = 0
hom((0, 1), (1, 0)) = 0
hom((0, 1), (0, 1)) = 1
hom((0, 1), (1, 1)) = 1
hom((1, 1), (1, 0)) = 1
hom((1, 1), (0, 1)) = 0
hom((1, 1), (1, 1)) = 0
```

canonical_stability_parameter(*d*)

Returns the canonical stability parameter for d

INPUT:

- d: dimension vector

OUTPUT: canonical stability parameter

The canonical stability parameter is given by *langle d,-rangle - langle -,drangle*.

INPUT:

- d – dimension vector to be used

OUTPUT: canonical stability parameter for d

EXAMPLES:

Our usual example of the 3-Kronecker quiver:

```
sage: from quiver import *
sage: Q = KroneckerQuiver(3)
sage: Q.canonical_stability_parameter((2, 3))
(9, -6)
```

For the 5-subspace quiver:

```
sage: Q = SubspaceQuiver(5)
sage: Q.canonical_stability_parameter((1, 1, 1, 1, 1, 2))
(2, 2, 2, 2, 2, -5)
```

It takes vertex labels (if present) into account:

```
sage: Q = Quiver.from_string("foo---bar", forget_labels=False)
sage: Q.canonical_stability_parameter((2, 3))
{'bar': -6, 'foo': 9}
```

EXAMPLES:

Canonical stability parameter for the 3-Kronecker quiver:

```
sage: from quiver import *
sage: Q, d = GeneralizedKroneckerQuiver(3), (2, 3)
sage: Q.canonical_stability_parameter(d)
(9, -6)
```

This method also works with vertex labels:

```
sage: from quiver import *
sage: Q = Quiver.from_string("foo---bar", forget_labels=False)
sage: d = {"foo": 2, "bar": 3}
sage: Q.canonical_stability_parameter(d)
{'bar': -6, 'foo': 9}
```

has_semistable_representation(*d*, *theta=None*, *denom=<built-in function sum>*)

Checks if there is a `theta`-semistable of dimension vector *d*

INPUT:

- `d`: dimension vector
- `theta` (default: canonical stability parameter): stability parameter

OUTPUT: whether there is a `theta`-semistable of dimension vector *d*

By MR1162487 a dimension vector *d* admits a θ-semi-stable representation if and only if $\mu_\theta(e) <= \mu_\theta(d)$ for all generic subdimension vectors *e* of *d*.

EXAMPLES:

Semistables for the A_2 quiver:

```
sage: from quiver import *
sage: Q = GeneralizedKroneckerQuiver(1)
sage: Q.has_semistable_representation((1, 1), (1, -1))
True
sage: Q.has_semistable_representation((2, 2), (1, -1))
True
sage: Q.has_semistable_representation((1, 2), (1, -1))
False
sage: Q.has_semistable_representation((0, 0), (1, -1))
True
```

Semistables for the 3-Kronecker quiver:

```
sage: from quiver import *
sage: Q = GeneralizedKroneckerQuiver(3)
sage: Q.has_semistable_representation((2, 3))
True
sage: Q.has_semistable_representation((1, 4), (-3, 2))
False
```

has_stable_representation(*d*, *theta=None*, *denom=<built-in function sum>*)

Checks if there is a `theta`-stable representation of this dimension vector

INPUT:

- `d`: dimension vector
- `theta` (default: canonical stability parameter): stability parameter

OUTPUT: whether there is a `theta`-stable of dimension vector `d`

By MR1162487 d admits a theta-stable representation if and only if $\mu_\theta(e) < \mu_\theta(d)$ for all proper generic subdimension vectors e of d.

EXAMPLES:

Stables for the A_2 quiver:

```
sage: from quiver import *
sage: Q = GeneralizedKroneckerQuiver(1)
sage: theta = (1, -1)
sage: Q.has_stable_representation((1, 1), theta)
True
sage: Q.has_stable_representation((2, 2), theta)
False
sage: Q.has_stable_representation((0, 0), theta)
False
```

Stables for the 3-Kronecker quiver:

```
sage: from quiver import *
sage: Q = GeneralizedKroneckerQuiver(3)
sage: d = (2, 3)
sage: theta = Q.canonical_stability_parameter(d)
sage: Q.has_stable_representation(d, theta)
True
sage: Q.has_stable_representation(d)
True
```

canonical_decomposition(*d*)

Computes the canonical decomposition of a dimension vector.

INPUT:

- `d`: dimension vector

OUTPUT: canonical decomposition as list of dimension vectors

The canonical decomposition of a dimension vector d is the unique decomposition $d = e_1 + e_2 + ... + e_k$ such that $e_1, e_2, ..., e_k$ are such that for all $i \neq j, \text{ext}(e_i, e_j) = \text{ext}(e_j, e_i) = 0$.

The general representation of dimension vector d is isomorphic to the direct sum of representations of dimension vectors $e_1, e_2, ..., e_k$.

EXAMPLES:

Canonical decomposition of the 3-Kronecker quiver:

```
sage: from quiver import *
sage: Q = GeneralizedKroneckerQuiver(3)
sage: Q.canonical_decomposition((2, 3))
[(2, 3)]
```

(continues on next page)

(continued from previous page)

```
sage: for d in Q.all_subdimension_vectors((5, 5)):
....:     print(Q.canonical_decomposition(d))
[(0, 0)]
[(0, 1)]
[(0, 1), (0, 1)]
[(0, 1), (0, 1), (0, 1)]
[(0, 1), (0, 1), (0, 1), (0, 1)]
[(0, 1), (0, 1), (0, 1), (0, 1), (0, 1)]
[(1, 0)]
[(1, 1)]
[(1, 2)]
[(1, 3)]
[(0, 1), (1, 3)]
[(0, 1), (0, 1), (1, 3)]
[(1, 0), (1, 0)]
[(2, 1)]
[(2, 2)]
[(2, 3)]
[(2, 4)]
[(2, 5)]
[(1, 0), (1, 0), (1, 0)]
[(3, 1)]
[(3, 2)]
[(3, 3)]
[(3, 4)]
[(3, 5)]
[(1, 0), (1, 0), (1, 0), (1, 0)]
[(1, 0), (3, 1)]
[(4, 2)]
[(4, 3)]
[(4, 4)]
[(4, 5)]
[(1, 0), (1, 0), (1, 0), (1, 0), (1, 0)]
[(1, 0), (1, 0), (3, 1)]
[(5, 2)]
[(5, 3)]
[(5, 4)]
[(5, 5)]
```

dimension_nullcone(*d*)

Returns the dimension of the nullcone

The nullcone is the set of all nilpotent representations.

INPUT:

- d – dimension vector

OUTPUT: dimension of the nullcone

EXAMPLES:

The usual example of the 3-Kronecker quiver:

```
sage: from quiver import *
sage: Q = KroneckerQuiver(3)
sage: Q.dimension_nullcone((2, 3))
18
```

__hash__ = None

__weakref__

list of weak references to the object (if defined)

first_hochschild_cohomology()

Compute the dimension of the first Hochschild cohomology

This uses the formula of Happel from Proposition 1.6 in MR1035222. One needs the quiver to be acyclic for this, otherwise it is not necessarily finite-dimensional.

EXAMPLES:

The first Hochschild cohomology of the m-th generalized Kronecker quiver is the dimension of PGL_{m+1}:

```
sage: from quiver import *
sage: GeneralizedKroneckerQuiver(3).first_hochschild_cohomology()
8
```

The first Hochschild cohomology vanishes if and only if the quiver is a tree:

```
sage: from quiver import *
sage: SubspaceQuiver(7).first_hochschild_cohomology()
0
```

CHAPTER

TWO

MODULI SPACES

class quiver.**QuiverModuli**(*Q*, *d*, *theta=None*, *denom=<built-in function sum>*, *condition='semistable'*)

__init__(*Q*, *d*, *theta=None*, *denom=<built-in function sum>*, *condition='semistable'*)

Constructor for an abstract quiver moduli space.

This base class contains everything that is common between - quiver moduli spaces, i.e., varieties - quiver moduli stacks

INPUT:

- `Q` – quiver
- `d` — dimension vector
- `theta` – stability parameter (default: canonical stability parameter)
- `denom` – denominator for slope stability (default: `sum`), needs to be effective on the simple roots
- `condition` – whether to include all semistables, or only stables (default: "semistable")

See *QuiverModuliSpace* and *QuiverModuliStack* for more details.

EXAMPLES:

We can instantiate an abstract quiver moduli space:

```
sage: from quiver import *
sage: Q = KroneckerQuiver(3)
sage: X = QuiverModuli(Q, (2, 3))
sage: X
abstract moduli of semistable representations, with
- Q = 3-Kronecker quiver
- d = (2, 3)
- θ = (9, -6)
```

It has functionality common to both varieties and stacks, i.e., when it really concerns something involving the representation variety:

```
sage: X.all_harder_narasimhan_types()
[((1, 0), (1, 1), (0, 2)),
 ((1, 0), (1, 2), (0, 1)),
 ((1, 0), (1, 3)),
 ((1, 1), (1, 2)),
 ((2, 0), (0, 3)),
 ((2, 1), (0, 2)),
 ((2, 2), (0, 1)),
 ((2, 3),)]
```

But things like dimension depend on whether we consider it as a variety or as a stack, and thus these are not implemented:

```
sage: X.dimension()
Traceback (most recent call last):
...
NotImplementedError
```

repr()

Give a shorthand string presentation for an abstract quiver moduli space.

EXAMPLES:

A Kronecker moduli space:

```
sage: from quiver import *
sage: Q = KroneckerQuiver(3)
sage: QuiverModuli(Q, (2, 3))
abstract moduli of semistable representations, with
- Q = 3-Kronecker quiver
- d = (2, 3)
- θ = (9, -6)
```

to_space()

Make the abstract quiver moduli a variety.

This is an explicit way of casting an abstract *QuiverModuli* to a *QuiverModuliSpace*.

EXAMPLES:

From an abstract quiver moduli to a space:

```
sage: from quiver import *
sage: Q = KroneckerQuiver(3)
sage: X = QuiverModuli(Q, (2, 3))
sage: X.to_space()
moduli space of semistable representations, with
- Q = 3-Kronecker quiver
- d = (2, 3)
- θ = (9, -6)
```

From a stack to a space:

```
sage: X = QuiverModuliStack(Q, (2, 3))
sage: X.to_space()
moduli space of semistable representations, with
- Q = 3-Kronecker quiver
- d = (2, 3)
- θ = (9, -6)
```

to_stack()

Make the abstract quiver moduli a stack.

This is an explicit way of casting an abstract *QuiverModuli* to a *QuiverModuliStack*.

EXAMPLES:

From an abstract quiver moduli to a stack:

```
sage: from quiver import *
sage: Q = KroneckerQuiver(3)
sage: X = QuiverModuli(Q, (2, 3))
sage: X.to_stack()
moduli stack of semistable representations, with
- Q = 3-Kronecker quiver
- d = (2, 3)
- θ = (9, -6)
```

From a space to a stack:

```
sage: X = QuiverModuliSpace(Q, (2, 3))
sage: X.to_stack()
moduli stack of semistable representations, with
- Q = 3-Kronecker quiver
- d = (2, 3)
- θ = (9, -6)
```

quiver()

Returns the quiver of the moduli space.

OUTPUT: the underlying quiver as an instance of the *Quiver* class

EXAMPLES:

The quiver of a Kronecker moduli space:

```
sage: from quiver import *
sage: Q = GeneralizedKroneckerQuiver(3)
sage: X = QuiverModuli(Q, (2, 3))
sage: Q == X.quiver()
True
```

dimension_vector()

Returns the dimension vector of the moduli space.

OUTPUT: the dimension vector

EXAMPLES:

The dimension vector of a Kronecker moduli space:

```
sage: from quiver import *
sage: Q = GeneralizedKroneckerQuiver(3)
sage: X = QuiverModuli(Q, (2, 3))
sage: X.dimension_vector()
(2, 3)
```

The dimension vector is stored in the same format as it was given:

```
sage: Q = Quiver.from_string("foo---bar", forget_labels=False)
sage: X = QuiverModuli(Q, {"foo": 2, "bar": 3})
sage: X.dimension_vector()
{'bar': 3, 'foo': 2}
```

stability_parameter()

Returns the stability parameter of the moduli space.

OUTPUT: the stability parameter

EXAMPLES:

The stability parameter of a Kronecker moduli space:

```
sage: from quiver import *
sage: Q = GeneralizedKroneckerQuiver(3)
sage: X = QuiverModuli(Q, (2, 3), (3, -2))
sage: X.stability_parameter()
(3, -2)
```

The stability parameter is stored in the same format as it was given:

```
sage: Q = Quiver.from_string("foo---bar", forget_labels=False)
sage: d, theta = {"foo": 2, "bar": 3}, {"foo": 3, "bar": -2}
sage: X = QuiverModuliSpace(Q, d, theta);
sage: X.stability_parameter()
{'bar': -2, 'foo': 3}
```

denominator()

Returns the denominator of the slope function μ_θ.

OUTPUT: the denominator as a function

EXAMPLES:

The 3-Kronecker quiver:

```
sage: from quiver import *
sage: Q = GeneralizedKroneckerQuiver(3)
sage: X = QuiverModuliSpace(Q, (2, 3))
sage: X.denominator()
<built-in function sum>
```

is_nonempty() → bool

Checks if the moduli space is nonempty.

OUTPUT: whether there exist stable/semistable representations, according to the condition

EXAMPLES:

The 3-Kronecker quiver for *d = (2, 3)* has stable representations:

```
sage: from quiver import *
sage: Q, d = GeneralizedKroneckerQuiver(3), (2, 3)
sage: X = QuiverModuliSpace(Q, d, condition="stable"); X.is_nonempty()
True
```

The Jordan quiver does not have stable representations, but it has semistable ones:

```
sage: Q = JordanQuiver()
sage: X = QuiverModuliSpace(Q, [3], condition="stable")
sage: X.is_nonempty()
False
sage: X = QuiverModuliSpace(Q, [3], condition="semistable")
sage: X.is_nonempty()
True
```

is_theta_coprime() → bool

Checks whether the combination of *d* and *theta* is coprime.

This just calls *Quiver.is_theta_coprime()* for the data defining the moduli space.

EXAMPLES:

A coprime example:

```
sage: from quiver import *
sage: Q = KroneckerQuiver(3)
sage: QuiverModuliSpace(Q, (2, 3)).is_theta_coprime()
True
```

And a non-example:

sage: QuiverModuliSpace(Q, (3, 3)).is_theta_coprime() False

all_harder_narasimhan_types (*proper=False*, *sorted=False*)

Returns the list of all Harder–Narasimhan types.

A Harder–Narasimhan (HN) type of d with respect to θ is a sequence $\mathbf{d}^* = (\mathbf{d}^1, ..., \mathbf{d}^s)$ of dimension vectors such that

- $\mathbf{d}^1 + ... + \mathbf{d}^s = \mathbf{d}$
- $\mu_\theta(\mathbf{d}^1) > ... > \mu_\theta(\mathbf{d}^s)$
- Every $\mathbf{d}^k$ is θ-semistable.

INPUT:

- `proper` – (default: False) whether to exclude the HN-type corresponding to the stable locus
- `sorted` – (default: False) whether to sort the HN-types according to the given slope

OUTPUT: list of tuples of dimension vectors encoding HN-types

EXAMPLES:

The 3-Kronecker quiver:

```
sage: from quiver import *
sage: Q = GeneralizedKroneckerQuiver(3)
sage: X = QuiverModuliSpace(Q, (2, 3))
sage: X.all_harder_narasimhan_types()
[((1, 0), (1, 1), (0, 2)),
 ((1, 0), (1, 2), (0, 1)),
 ((1, 0), (1, 3)),
 ((1, 1), (1, 2)),
 ((2, 0), (0, 3)),
 ((2, 1), (0, 2)),
 ((2, 2), (0, 1)),
 ((2, 3),)]
sage: X.all_harder_narasimhan_types(proper=True)
[((1, 0), (1, 1), (0, 2)),
 ((1, 0), (1, 2), (0, 1)),
 ((1, 0), (1, 3)),
 ((1, 1), (1, 2)),
 ((2, 0), (0, 3)),
 ((2, 1), (0, 2)),
 ((2, 2), (0, 1))]
sage: d = (2, 3)
sage: theta = -Q.canonical_stability_parameter(d)
sage: Y = QuiverModuliSpace(Q, d, theta)
sage: Y.all_harder_narasimhan_types()
[((0, 3), (2, 0))]
```

A 3-vertex quiver:

```
sage: from quiver import *
sage: Q = ThreeVertexQuiver(2, 3, 4)
sage: Z = QuiverModuliSpace(Q, (2, 3, 2))
sage: Z.all_harder_narasimhan_types()
[((0, 1, 0), (1, 2, 1), (1, 0, 1)),
 ((0, 1, 0), (2, 0, 1), (0, 2, 1)),
 ((0, 1, 0), (2, 1, 1), (0, 1, 1)),
 ((0, 1, 0), (2, 2, 1), (0, 0, 1)),
 ((0, 1, 0), (2, 2, 2)),
 ((0, 2, 0), (1, 1, 1), (1, 0, 1)),
 ((0, 2, 0), (2, 0, 1), (0, 1, 1)),
 ((0, 2, 0), (2, 1, 1), (0, 0, 1)),
 ((0, 2, 0), (2, 1, 2)),
 ((0, 3, 0), (2, 0, 1), (0, 0, 1)),
 ((0, 3, 0), (2, 0, 2)),
 ((1, 0, 0), (0, 1, 0), (1, 0, 1), (0, 2, 1)),
 ((1, 0, 0), (0, 1, 0), (1, 1, 1), (0, 1, 1)),
 ((1, 0, 0), (0, 1, 0), (1, 2, 1), (0, 0, 1)),
 ((1, 0, 0), (0, 1, 0), (1, 2, 2)),
 ((1, 0, 0), (0, 2, 0), (1, 0, 1), (0, 1, 1)),
 ((1, 0, 0), (0, 2, 0), (1, 1, 1), (0, 0, 1)),
 ((1, 0, 0), (0, 2, 0), (1, 1, 2)),
 ((1, 0, 0), (0, 3, 0), (1, 0, 1), (0, 0, 1)),
 ((1, 0, 0), (0, 3, 0), (1, 0, 2)),
 ((1, 0, 0), (0, 3, 1), (1, 0, 1)),
 ((1, 0, 0), (1, 1, 0), (0, 1, 0), (0, 1, 1), (0, 0, 1)),
 ((1, 0, 0), (1, 1, 0), (0, 1, 0), (0, 1, 2)),
 ((1, 0, 0), (1, 1, 0), (0, 2, 0), (0, 0, 2)),
 ((1, 0, 0), (1, 1, 0), (0, 2, 1), (0, 0, 1)),
 ((1, 0, 0), (1, 1, 0), (0, 2, 2)),
 ((1, 0, 0), (1, 1, 1), (0, 2, 1)),
 ((1, 0, 0), (1, 2, 0), (0, 1, 0), (0, 0, 2)),
 ((1, 0, 0), (1, 2, 0), (0, 1, 1), (0, 0, 1)),
 ((1, 0, 0), (1, 2, 0), (0, 1, 2)),
 ((1, 0, 0), (1, 2, 1), (0, 1, 1)),
 ((1, 0, 0), (1, 3, 1), (0, 0, 1)),
 ((1, 0, 0), (1, 3, 2)),
 ((1, 1, 0), (0, 1, 0), (1, 0, 1), (0, 1, 1)),
 ((1, 1, 0), (0, 1, 0), (1, 1, 1), (0, 0, 1)),
 ((1, 1, 0), (0, 1, 0), (1, 1, 2)),
 ((1, 1, 0), (0, 2, 0), (1, 0, 1), (0, 0, 1)),
 ((1, 1, 0), (0, 2, 0), (1, 0, 2)),
 ((1, 1, 0), (1, 0, 1), (0, 2, 1)),
 ((1, 1, 0), (1, 1, 1), (0, 1, 1)),
 ((1, 1, 0), (1, 2, 0), (0, 0, 2)),
 ((1, 1, 0), (1, 2, 1), (0, 0, 1)),
 ((1, 1, 0), (1, 2, 2)),
 ((1, 2, 0), (0, 1, 0), (1, 0, 1), (0, 0, 1)),
 ((1, 2, 0), (0, 1, 0), (1, 0, 2)),
 ((1, 2, 0), (1, 0, 1), (0, 1, 1)),
 ((1, 2, 0), (1, 1, 1), (0, 0, 1)),
 ((1, 2, 0), (1, 1, 2)),
 ((1, 2, 1), (1, 1, 1)),
 ((1, 3, 1), (1, 0, 1)),
 ((2, 0, 0), (0, 1, 0), (0, 2, 1), (0, 0, 1)),
 ((2, 0, 0), (0, 1, 0), (0, 2, 2)),
```

(continues on next page)

(continued from previous page)

```
 ((2, 0, 0), (0, 2, 0), (0, 1, 1), (0, 0, 1)),
 ((2, 0, 0), (0, 2, 0), (0, 1, 2)),
 ((2, 0, 0), (0, 2, 1), (0, 1, 1)),
 ((2, 0, 0), (0, 3, 0), (0, 0, 2)),
 ((2, 0, 0), (0, 3, 1), (0, 0, 1)),
 ((2, 0, 0), (0, 3, 2)),
 ((2, 0, 1), (0, 3, 1)),
 ((2, 1, 0), (0, 1, 0), (0, 1, 1), (0, 0, 1)),
 ((2, 1, 0), (0, 1, 0), (0, 1, 2)),
 ((2, 1, 0), (0, 2, 0), (0, 0, 2)),
 ((2, 1, 0), (0, 2, 1), (0, 0, 1)),
 ((2, 1, 0), (0, 2, 2)),
 ((2, 1, 1), (0, 2, 1)),
 ((2, 2, 0), (0, 1, 0), (0, 0, 2)),
 ((2, 2, 0), (0, 1, 1), (0, 0, 1)),
 ((2, 2, 0), (0, 1, 2)),
 ((2, 2, 1), (0, 1, 1)),
 ((2, 3, 0), (0, 0, 2)),
 ((2, 3, 1), (0, 0, 1)),
 ((2, 3, 2),)]
```

is_harder_narasimhan_type(*dstar*) → bool

Checks if `dstar` is a Harder–Narasimhan type.

A Harder–Narasimhan (HN) type of d with respect to θ is a sequence $\mathbf{d}^* = (\mathbf{d}^1, ..., \mathbf{d}^s)$ of dimension vectors such that

- $\mathbf{d}^1 + ... + \mathbf{d}^s = \mathbf{d}$
- $\mu_\theta(\mathbf{d}^1) > ... > \mu_\theta(\mathbf{d}^s)$
- Every $\mathbf{d}^k$ is θ-semistable.

INPUT:

- `dstar` – list of dimension vectors

OUTPUT: whether `dstar` is a valid HN type for the moduli space

EXAMPLES:

The 3-Kronecker quiver:

```
sage: from quiver import *
sage: Q = GeneralizedKroneckerQuiver(3)
sage: X = QuiverModuliSpace(Q, (2, 3))
sage: HNs = X.all_harder_narasimhan_types()
sage: all(X.is_harder_narasimhan_type(dstar) for dstar in HNs)
True
sage: dstar = [(1, 0), (1, 0), (0, 3)]
sage: X.is_harder_narasimhan_type(dstar)
False
sage: X.is_harder_narasimhan_type([Q.zero_vector()])
False
```

codimension_of_harder_narasimhan_stratum(*dstar*, *secure=False*)

Computes the codimension of the HN stratum of `dstar` inside the representation variety $R(Q, d)$.

INPUT:

- `dstar` – the HN type as a list of dimension vectors

- `secure` – whether to first check it is an HN-type (default: False)

OUTPUT: codimension as an integer

By default, the method does not check if `dstar` is a valid HN type. This can be enabled by passing `secure=True`.

The codimension of the HN stratum of $d^* = (d^1, ..., d^s)$ is given by

$$-\sum_{k<l}\langle \mathbf{d}^k, \mathbf{d}^l\rangle$$

INPUT:

- `dstar` – list of dimension vectors
- `secure` – whether to check `dstar` is an HN-type (default: False)

OUTPUT: codimension of the HN-stratum

EXAMPLES:

The 3-Kronecker quiver:

```
sage: from quiver import *
sage: Q = GeneralizedKroneckerQuiver(3)
sage: X = QuiverModuliSpace(Q, (2, 3))
sage: HNs = X.all_harder_narasimhan_types()
sage: [X.codimension_of_harder_narasimhan_stratum(dstar) for dstar in HNs]
[12, 9, 8, 3, 18, 10, 4, 0]
```

codimension_unstable_locus()

Computes codimension of the unstable locus inside the representation variety.

This is the minimum of the codimensions of the proper Harder–Narasimhan strata of the representation variety.

OUTPUT: codimension of the unstable locus

EXAMPLES:

The 3-Kronecker quiver:

```
sage: from quiver import *
sage: Q = GeneralizedKroneckerQuiver(3)
sage: X = QuiverModuliSpace(Q, (2, 3))
sage: X.codimension_unstable_locus()
3
```

A 3-vertex quiver:

```
sage: Q = ThreeVertexQuiver(1, 6, 1)
sage: X = QuiverModuliSpace(Q, (1, 6, 6))
sage: X.codimension_unstable_locus()
1
```

The A_2 quiver is of finite type:

```
sage: Q = GeneralizedKroneckerQuiver(1)
sage: X = QuiverModuliSpace(Q, (2, 3))
sage: X.codimension_unstable_locus()
0
```

all_luna_types(*exclude_stable=False*)

Returns the unordered list of all Luna types of `d` for `theta`.

INPUT:

- `exclude_stable` – whether to exclude the stable Luna type `{d: [1]}` (default: False)

OUTPUT: the list of all the Luna types as dictionaries.

The Luna stratification of the representation variety concerns the étale-local structure of the moduli space of semistable quiver representations. It is studied in MR1972892, and for more details one is referred there.

A Luna type of $\mathbf{d}$ for θ is an unordered sequence $((\mathbf{d}^1, m_1), ..., (\mathbf{d}^s, m_s))$ of pairs of dimension vectors $\mathbf{d}^k$ and positive integers m_k such that

- $m_1\mathbf{d}^1 + ... + m_s\mathbf{d}^s = \mathbf{d}$,
- $\mu_\theta(\mathbf{d}^k) = \mu_\theta(\mathbf{d})$, and
- all the $\mathbf{d}^k$ admit a θ-stable representation.

Note that a pair $(\mathbf{d}^i, m_i)$ can appear multiple times in a Luna type, and the same dimension vector $\mathbf{d}^i$ can appear coupled with different integers.

IMPLEMENTATION:

Here a Luna type is a dictionary `{d^1: p^1, ... d^s: p^s}` whose keys are dimension vectors $\mathbf{d}^k$ and values are non-empty lists of positive integers `p^k = [p_{k, 1}, ..., p_{k, t_k}]`.

The corresponding Luna type is then the unordered sequence of tuples

$$(\mathbf{d}^1, p_{1,1}), \ldots, (\mathbf{d}^1, p_{1,t_1}), \ldots (\mathbf{d}^s, p_{s,1}), \ldots, (\mathbf{d}^s, p_{s,t_s}),$$

such that

$$(p_{1,1} + \cdots + p_{1,t_1}) \cdot \mathbf{d}^1 + \cdots + (p_{s,1} + \cdots + p_{s,t_s}) \cdot \mathbf{d}^s = \mathbf{d}.$$

ALGORITHM:

The way we compute the Luna types of a quiver moduli space is taken from Section 4 in MR2511752.

EXAMPLES:

The Kronecker quiver:

```
sage: from quiver import *
sage: Q = KroneckerQuiver()
sage: X = QuiverModuliSpace(Q, (3, 3), (1, -1))
sage: X.all_luna_types()
[{(1, 1): [3]}, {(1, 1): [2, 1]}, {(1, 1): [1, 1, 1]}]
```

The 3-Kronecker quiver:

```
sage: from quiver import *
sage: Q = GeneralizedKroneckerQuiver(3)
sage: X = QuiverModuliSpace(Q, (3, 3), (1, -1))
sage: X.all_luna_types()
[{(3, 3): [1]},
 {(1, 1): [1], (2, 2): [1]},
 {(1, 1): [3]},
 {(1, 1): [2, 1]},
 {(1, 1): [1, 1, 1]}]
```

The zero vector:

```
sage: from quiver import *
sage: Q = KroneckerQuiver()
sage: X = QuiverModuliSpace(Q, (0, 0), (1, -1))
sage: X.all_luna_types()
[{(0, 0): [1]}]
```

is_luna_type(*tau*) → bool

Checks if `tau` is a Luna type.

A Luna type of $\mathbf{d}$ for θ is an unordered sequence $((\mathbf{d}^1, m_1), ..., (\mathbf{d}^s, m_s))$ of dimension vectors $\mathbf{d}^k$ and positive integers m_k such that

- $m_1\mathbf{d}^1 + ... + m_s\mathbf{d}^s = \mathbf{d}$
- $\mu_\theta(\mathbf{d}^k) = \mu_\theta(\mathbf{d})$
- All $\mathbf{d}^k$ admit a θ-stable representation

INPUT:

- `tau` – Luna type encoded by a dictionary of multiplicities indexed by dimension vectors

OUTPUT: whether `tau` is a Luna type

EXAMPLES:

The Kronecker quiver:

```
sage: from quiver import *
sage: Q = KroneckerQuiver()
sage: X = QuiverModuliSpace(Q, (3, 3), (1, -1))
sage: Ls = X.all_luna_types()
sage: all(X.is_luna_type(tau) for tau in Ls)
True
```

The 3-Kronecker quiver with zero vector:

```
sage: from quiver import *
sage: Q = KroneckerQuiver()
sage: X = QuiverModuliSpace(Q, (0, 0), (1, -1))
sage: X.is_luna_type({Q.zero_vector(): [1]})
True
```

dimension_of_luna_stratum(*tau*, *secure=True*)

Computes the dimension of the Luna stratum S_τ.

INPUT:

- `tau` – Luna type encoded by a dictionary of multiplicities indexed by dimension vectors
- `secure` – whether to first check it is a Luna type (default: False)

OUTPUT: dimension of the corresponding Luna stratum

The dimension of the Luna stratum of `tau = {d^1: p^1,...,d^s: p^s}` is

$$\sum_k l(p^k)(1 - \langle \mathbf{d}^k, \mathbf{d}^k \rangle)$$

where for a partition $p = (n_1, ..., n_l)$, the length *l(p)* is *l*, i.e., the number of summands.

EXAMPLES:

The Kronecker quiver:

```
sage: from quiver import *
sage: Q = KroneckerQuiver()
sage: X = QuiverModuliSpace(Q, (2, 2), (1, -1))
sage: Ls = X.all_luna_types(); Ls
[{(1, 1): [2]}, {(1, 1): [1, 1]}]
sage: [X.dimension_of_luna_stratum(tau) for tau in Ls]
[1, 2]
```

local_quiver_setting(*tau*, *secure=True*)

Returns the local quiver and dimension vector for the given Luna type.

The local quiver describes the singularities of a moduli space, and is introduced and studied in studied in MR1972892.

INPUT:

- `tau` – Luna type encoded by a dictionary of multiplicities indexed by dimension vectors
- `secure` – whether to first check it is a Luna type (default: False)

OUTPUT: tuple consisting of a Quiver object and a dimension vector

EXAMPLES:

The 3-Kronecker quiver:

```
sage: from quiver import *
sage: Q = GeneralizedKroneckerQuiver(3)
sage: X = QuiverModuliSpace(Q, (2, 2), (1, -1))
sage: Ls = X.all_luna_types(); Ls
[{(2, 2): [1]}, {(1, 1): [2]}, {(1, 1): [1, 1]}]
sage: Qloc, dloc = X.local_quiver_setting(Ls[0]);
sage: Qloc.adjacency_matrix(), dloc
([4], (1))
sage: Qloc, dloc = X.local_quiver_setting(Ls[1]);
sage: Qloc.adjacency_matrix(), dloc
([1], (2))
sage: Qloc, dloc = X.local_quiver_setting(Ls[2]);
sage: Qloc.adjacency_matrix(), dloc
(
[1 1]
[1 1], (1, 1)
)
```

codimension_properly_semistable_locus()

Computes the codimension of $R^{\theta-\mathrm{sst}}(Q,d) \setminus R^{\theta-\mathrm{st}}(Q,d)$ inside $R(Q,d)$.

OUTPUT: codimension of the properly semistable locus

The codimension of the properly semistable locus is the minimal codimension of the inverse image of the non-stable Luna strata.

EXAMPLES:

If the semistable locus is the stable locus the codimension is -Infinity:

```
sage: from quiver import *
sage: Q = KroneckerQuiver(3)
```

(continues on next page)

(continued from previous page)

```
sage: QuiverModuliSpace(Q, (2, 3)).codimension_properly_semistable_locus()
-Infinity
```

This is currently not working properly because we cannot compute the dimension of the nullcone:

```
sage: from quiver import *
sage: Q = KroneckerQuiver(3)
sage: QuiverModuliSpace(Q, (3, 3)).codimension_properly_semistable_locus()
Traceback (most recent call last):
...
NotImplementedError
```

semistable_equals_stable()

Checks whether every semistable representation is stable for the given stability parameter.

Every θ-semistable representation is θ-stable if and only if there are no Luna types other than (possibly) `{d: [1]}`.

OUTPUT: whether every theta-semistable representation is θ-stable

EXAMPLES:

The 3-Kronecker quiver:

```
sage: from quiver import *
sage: Q = GeneralizedKroneckerQuiver(3)
sage: X = QuiverModuliSpace(Q, (3, 3))
sage: X.semistable_equals_stable()
False
sage: Y = QuiverModuliSpace(Q, (2, 3))
sage: Y.semistable_equals_stable()
True
```

A double framed example as in our vector fields paper:

```
sage: from quiver import *
sage: Q = GeneralizedKroneckerQuiver(3)
sage: Q = Q.framed_quiver((1, 0)).coframed_quiver((0, 0, 1))
sage: d = (1, 2, 3, 1)
sage: theta = (1, 300, -200, -1)
sage: X = QuiverModuliSpace(Q, d, theta)
sage: X.is_theta_coprime()
False
sage: X.semistable_equals_stable()
True
```

is_amply_stable() → bool

Checks if the dimension vector is amply stable for the stability parameter

By definition, a dimension vector d is θ-amply stable if the codimension of the θ-semistable locus inside *R(Q,d)* is at least 2.

OUTPUT: whether the data for the quiver moduli space is amply stable

EXAMPLES:

3-Kronecker quiver:

```
sage: from quiver import *
sage: Q = GeneralizedKroneckerQuiver(3)
sage: QuiverModuliSpace(Q, (2, 3)).is_amply_stable()
True
sage: QuiverModuliSpace(Q, (2, 3), [-3, 2]).is_amply_stable()
False
```

A three-vertex example from the rigidity paper:

```
sage: Q = ThreeVertexQuiver(1, 6, 1)
sage: QuiverModuliSpace(Q, [1, 6, 6]).is_amply_stable()
False
```

is_strongly_amply_stable() → bool

Checks if the dimension vector is strongly amply stable for the stability parameter

We call $\mathbf{d}$ strongly amply stable for θ if $\langle \mathbf{e}, \mathbf{d} - \mathbf{e} \rangle \leq -2$ holds for all subdimension vectors $\mathbf{e}$ of $\mathbf{d}$ for which $\mu_\theta(\mathbf{e}) \geq \mu_\theta(\mathbf{d})$.

OUTPUT: whether the data for the quiver moduli space is strongly amply stable

EXAMPLES:

3-Kronecker quiver:

```
sage: from quiver import *
sage: Q = GeneralizedKroneckerQuiver(3)
sage: QuiverModuliSpace(Q, (2, 3)).is_strongly_amply_stable()
True
```

A 3-vertex quiver:

```
sage: from quiver import *
sage: Q = ThreeVertexQuiver(5, 1, 1)
sage: X = QuiverModuliSpace(Q, [4, 1, 4])
sage: X.is_amply_stable()
True
sage: X.is_strongly_amply_stable()
False
```

harder_narasimhan_weight(*harder_narasimhan_type*)

Returns the Teleman weight of a Harder-Narasimhan type

INPUT:

- harder_narasimhan_type – list of vectors of Ints

OUTPUT: weight as a fraction

The weight of a Harder-Narasimhan type d^* is the weight of the associated 1-PS λ acting on $\det(N_{S/R})^\vee|_Z$, where S is the corresponding Harder–Narasimhan stratum.

See also:

all_weight_bounds(), *if_rigidity_inequality_holds()*

EXAMPLES:

The 3-Kronecker quiver:

```
sage: from quiver import *
sage: Q = GeneralizedKroneckerQuiver(3)
sage: X = QuiverModuliSpace(Q, (2, 3))
sage: HN = X.all_harder_narasimhan_types(proper=True)
sage: {dstar: X.harder_narasimhan_weight(dstar) for dstar in HN}
{((1, 0), (1, 1), (0, 2)): 135,
 ((1, 0), (1, 2), (0, 1)): 100,
 ((1, 0), (1, 3)): 90,
 ((1, 1), (1, 2)): 15/2,
 ((2, 0), (0, 3)): 270,
 ((2, 1), (0, 2)): 100,
 ((2, 2), (0, 1)): 30}
```

all_weight_bounds(*as_dict=False*)

Returns the list of all weights appearing in Teleman quantization.

For each HN type, the 1-PS lambda acts on $\det(N^\vee_{S/R}|_Z)$ with a certain weight. Teleman quantization gives a numerical condition involving these weights to compute cohomology on the quotient.

INPUT:

- `as_dict` – (default: False) when True it will give a dict whose keys are the HN-types and whose values are the weights

EXAMPLES:

The 6-dimensional 3-Kronecker example:

```
sage: from quiver import *
sage: X = QuiverModuliSpace(KroneckerQuiver(3), (2, 3))
sage: X.all_weight_bounds()
[135, 100, 90, 15/2, 270, 100, 30]
sage: X.all_weight_bounds(as_dict=True)
{((1, 0), (1, 1), (0, 2)): 135,
 ((1, 0), (1, 2), (0, 1)): 100,
 ((1, 0), (1, 3)): 90,
 ((1, 1), (1, 2)): 15/2,
 ((2, 0), (0, 3)): 270,
 ((2, 1), (0, 2)): 100,
 ((2, 2), (0, 1)): 30}
```

if_rigidity_inequality_holds() → bool

OUTPUT: whether the rigidity inequality holds on the given moduli

If the weights of the 1-PS lambda on $\det(N_{S/R}|_Z)$ for each HN type are all strictly larger than the weights of the tensors of the universal bundles $U_i^\vee \otimes U_j$, then the resulting moduli space is infinitesimally rigid.

EXAMPLES:

Kronecker moduli satisfy the rigidity inequality:

```
sage: from quiver import *
sage: X = QuiverModuliSpace(KroneckerQuiver(3), (2, 3))
sage: X.if_rigidity_inequality_holds()
True
```

The following 3-vertex example does not (however, it is rigid by other means):

```
sage: X = QuiverModuliSpace(ThreeVertexQuiver(1, 6, 1), [1, 6, 6])
sage: X.if_rigidity_inequality_holds()
False
```

tautological_ideal(*use_roots=False*, *classes=None*, *roots=None*)

Returns the tautological presentation of the Chow ring of the moduli space.

INPUT:

- `use_roots` – (default: False) whether to return the relations in Chern roots
- `classes` – (default: None) optional list of strings to name the Chern classes
- `roots` – (default: None) optional list of strings to name the Chern roots

OUTPUT: ideal of a polynomial ring

EXAMPLES:

The tautological ideal for our favourite 6-fold has 9 non-zero generators:

```
sage: from quiver import *
sage: Q = GeneralizedKroneckerQuiver(3)
sage: X = QuiverModuliSpace(Q, (2, 3))
sage: len(X.tautological_ideal().gens())
9
```

dimension() → int

Returns the dimension of the moduli stack.

See also:

- *QuiverModuliSpace.dimension()*
- *QuiverModuliStack.dimension()*

EXAMPLES:

This is not implemented as it is ambiguous:

```
sage: from quiver import *
sage: Q = KroneckerQuiver(3)
sage: QuiverModuli(Q, (2, 3)).dimension()
Traceback (most recent call last):
...
NotImplementedError
```

is_smooth() → bool

Checks if the moduli space is smooth.

Abstract method, see the concrete implementations for details.

See also:

- *QuiverModuliSpace.is_smooth()*
- *QuiverModuliStack.is_smooth()*

This is not implemented as it is ambiguous:

```
sage: from quiver import *
sage: Q = KroneckerQuiver(3)
sage: QuiverModuli(Q, (2, 3)).is_smooth()
Traceback (most recent call last):
...
NotImplementedError
```

chow_ring()

Returns the Chow ring of the moduli space.

Abstract method, see the concrete implementations for details.

See also:

- *QuiverModuliSpace.chow_ring()*
- *QuiverModuliStack.chow_ring()*

EXAMPLES:

This is not implemented as it is ambiguous:

```
sage: from quiver import *
sage: Q = KroneckerQuiver(3)
sage: QuiverModuli(Q, (2, 3)).is_smooth()
Traceback (most recent call last):
...
NotImplementedError
```

__weakref__

list of weak references to the object (if defined)

class quiver.**QuiverModuliSpace**(*Q*, *d*, *theta=None*, *denom=<built-in function sum>*, *condition='semistable'*)

__init__(*Q*, *d*, *theta=None*, *denom=<built-in function sum>*, *condition='semistable'*)

Constructor for a quiver moduli space

This is the quiver moduli space as a variety.

INPUT:

- `Q` – quiver
- `d` — dimension vector
- `theta` – stability parameter (default: canonical stability parameter)
- `denom` – denominator for slope stability (default: `sum`), needs to be effective on the simple roots
- `condition` – whether to include all semistables, or only stables (default: “semistable”)

EXAMPLES:

An example:

```
sage: from quiver import *
sage: Q = KroneckerQuiver(3)
sage: QuiverModuliSpace(Q, (2, 3))
moduli space of semistable representations, with
- Q = 3-Kronecker quiver
```

(continues on next page)

(continued from previous page)

```
- d = (2, 3)
- θ = (9, -6)
```

repr()

Give a shorthand string presentation for the quiver moduli space

EXAMPLES:

A Kronecker moduli space:

```
sage: from quiver import *
sage: Q = KroneckerQuiver(3)
sage: QuiverModuliSpace(Q, (2, 3))
moduli space of semistable representations, with
- Q = 3-Kronecker quiver
- d = (2, 3)
- θ = (9, -6)
```

dimension()

Computes the dimension of the moduli space $M^{\theta-(s)st}(Q,d)$.

This involves several cases:

- If there are θ-stable representations then $\dim M^{\theta-\mathrm{sst}}(Q,d) = M^{\theta-st}(Q,d) = 1 - \langle d,d\rangle$;
- if there are no θ-stable representations then $\dim M^{\theta-st}(Q,d) = -\infty$ by convention, and we define $\dim M^{\theta-\mathrm{sst}} = \max_\tau\{\dim S_\tau\}$, the maximum of the dimension of all Luna strata.

EXAMPLES

The A2-quiver:

```
sage: from quiver import *
sage: Q = GeneralizedKroneckerQuiver(1)
sage: X = QuiverModuliSpace(Q, [1, 1], condition="stable")
sage: X.dimension()
0
sage: X = QuiverModuliSpace(Q, [1, 1], condition="semistable")
sage: X.dimension()
0
sage: X = QuiverModuliSpace(Q, [2, 2], condition="stable")
sage: X.dimension()
-Infinity
sage: X = QuiverModuliSpace(Q, [2, 2], condition="semistable")
sage: X.dimension()
0
```

The Kronecker quiver:

```
sage: from quiver import *
sage: Q = GeneralizedKroneckerQuiver(2)
sage: X = QuiverModuliSpace(Q, [1, 1], [1, -1], condition="stable")
sage: X.dimension()
1
sage: X = QuiverModuliSpace(Q, [1, 1], [1, -1], condition="semistable")
sage: X.dimension()
1
sage: X = QuiverModuliSpace(Q, [2, 2], [1, -1], condition="stable")
```

(continues on next page)

(continued from previous page)

```
sage: X.dimension()
-Infinity
sage: X = QuiverModuliSpace(Q, [2, 2], [1, -1], condition="semistable")
sage: X.dimension()
2
```

The 3-Kronecker quiver:

```
sage: from quiver import *
sage: Q = GeneralizedKroneckerQuiver(3)
sage: X = QuiverModuliSpace(Q, (2, 3), condition="semistable")
sage: X.dimension()
6
sage: X = QuiverModuliSpace(Q, [3, 3],condition="semistable")
sage: X.dimension()
10
sage: X = QuiverModuliSpace(Q, [1, 3],condition="stable")
sage: X.dimension()
0
sage: X = QuiverModuliSpace(Q, [1, 4],condition="stable")
sage: X.dimension()
-Infinity
sage: X = QuiverModuliSpace(Q, [1, 4],condition="semistable")
sage: X.dimension()
-Infinity
```

The Jordan quiver:

```
sage: QuiverModuliSpace(JordanQuiver(1), (0,)).dimension()
0
sage: X = QuiverModuliSpace(JordanQuiver(1), (0,), condition="stable")
sage: X.dimension()
-Infinity
sage: QuiverModuliSpace(JordanQuiver(1), (1,)).dimension()
1
sage: QuiverModuliSpace(JordanQuiver(1), (2,)).dimension()
2
sage: QuiverModuliSpace(JordanQuiver(1), (3,)).dimension()
3
sage: QuiverModuliSpace(JordanQuiver(1), (4,)).dimension()
4
```

Some generalized Jordan quivers:

```
sage: QuiverModuliSpace(JordanQuiver(2), (0,)).dimension()
0
sage: QuiverModuliSpace(JordanQuiver(2), (1,)).dimension()
2
sage: QuiverModuliSpace(JordanQuiver(2), (2,)).dimension()
5
sage: QuiverModuliSpace(JordanQuiver(2), (3,)).dimension()
10
sage: QuiverModuliSpace(JordanQuiver(2), (4,)).dimension()
17
```

More generalized Jordan quivers:

```
sage: QuiverModuliSpace(JordanQuiver(3), (0,)).dimension()
0
sage: QuiverModuliSpace(JordanQuiver(3), (1,)).dimension()
3
sage: QuiverModuliSpace(JordanQuiver(3), (2,)).dimension()
9
sage: QuiverModuliSpace(JordanQuiver(3), (3,)).dimension()
19
sage: QuiverModuliSpace(JordanQuiver(3), (4,)).dimension()
33
```

poincare_polynomial()

Returns the Poincare polynomial of the moduli space.

OUTPUT: Poincaré polynomial in the variable `q`

The Poincare polynomial is defined as

$$P_X(q) = \sum_{i \geq 0} (-1)^i \dim \mathrm{H}^i(X; \mathbb{C}) q^{i/2}$$

For a quiver moduli space whose dimension vector is θ-coprime, the odd cohomology vanishes and this is a polynomial in q.

ALGORITHM:

Corollary 6.9 in MR1974891.

EXAMPLES:

Some Kronecker quivers:

```
sage: from quiver import *
sage: Q = KroneckerQuiver()
sage: X = QuiverModuliSpace(Q, (1, 1))
sage: X.poincare_polynomial()
q + 1
sage: Q = GeneralizedKroneckerQuiver(3)
sage: X = QuiverModuliSpace(Q, (2, 3))
sage: X.poincare_polynomial()
q^6 + q^5 + 3*q^4 + 3*q^3 + 3*q^2 + q + 1
sage: Q = SubspaceQuiver(5)
sage: X = QuiverModuliSpace(Q, (1, 1, 1, 1, 1, 2))
sage: X.poincare_polynomial()
q^2 + 5*q + 1
```

betti_numbers()

Returns the Betti numbers of the moduli space.

OUTPUT: Betti numbers of the moduli space

ALGORITHM:

Corollary 6.9 in MR1974891.

EXAMPLES:

Some Kronecker quivers:

```
sage: from quiver import *
sage: Q = KroneckerQuiver()
sage: X = QuiverModuliSpace(Q, (1, 1), condition="semistable")
sage: X.poincare_polynomial()
q + 1
sage: X.betti_numbers()
[1, 0, 1]
sage: Q = GeneralizedKroneckerQuiver(3)
sage: X = QuiverModuliSpace(Q, (2, 3), condition="semistable")
sage: X.betti_numbers()
[1, 0, 1, 0, 3, 0, 3, 0, 3, 0, 1, 0, 1]
```

is_smooth() → bool

Returns whether the moduli space is smooth.

This is easy if the condition is "stable", because this moduli space is always smooth. In the "semistable" case there is an algorithm, by combining the work of Adriaenssens–Le Bruyn and Bocklandt, which is currently not implemented.

EXAMPLES:

Some 3-Kronecker example:

```
sage: from quiver import *
sage: Q = KroneckerQuiver(3)
sage: QuiverModuliSpace(Q, (2, 3)).is_smooth()
True
sage: QuiverModuliSpace(Q, (2, 3), condition="stable").is_smooth()
True
sage: QuiverModuliSpace(Q, (3, 3), condition="stable").is_smooth()
True
sage: QuiverModuliSpace(Q, (3, 3)).is_smooth()
Traceback (most recent call last):
...
NotImplementedError
```

semisimple_moduli_space()

Return the moduli space with *theta* replaced by zero.

This is the moduli space of semisimple representations for the same quiver and the same dimension vector.

EXAMPLES:

For an acyclic quiver this moduli space is a point:

```
sage: from quiver import *
sage: Q = KroneckerQuiver(3)
sage: X = QuiverModuliSpace(Q, (2, 3))
sage: X.semisimple_moduli_space().dimension()
0
```

For a quiver with oriented cycles we get an affine variety:

```
sage: Q = JordanQuiver(2)
sage: X = QuiverModuliSpace(Q, (3,))
sage: X.dimension()
10
```

is_projective() → bool

Check whether the moduli space is projective

EXAMPLES:

For acyclic quivers the semistable moduli space is always projective:

```
sage: from quiver import *
sage: Q = KroneckerQuiver(3)
sage: QuiverModuliSpace(Q, (2, 3)).is_projective()
True
```

If we have strictly semistable representations, then the stable moduli space is only quasiprojective but not projective:

```
sage: QuiverModuliSpace(Q, (3, 3), condition="stable").is_projective()
False
```

In pathological cases we can have that the affine moduli space of semisimples is reduced to a point, and the projective-over-affine becomes projective:

```
sage: Q = CyclicQuiver(3)
sage: QuiverModuliSpace(Q, (2, 0, 2)).is_projective()
True
```

For the zero dimension vector we get either a point or an empty space, which is always projective:

```
sage: Q = KroneckerQuiver(3)
sage: QuiverModuliSpace(Q, (0, 0)).is_projective()
True
sage: QuiverModuliSpace(Q, (0, 0), condition="stable").is_projective()
True
```

picard_rank()

Computes the Picard rank of the moduli space.

We compute this as the Betti number b_2.

EXAMPLES:

Kronecker moduli are rank 1:

```
sage: from quiver import *
sage: Q = KroneckerQuiver(3)
sage: QuiverModuliSpace(Q, (2, 3)).picard_rank()
1
```

index()

Computes the index of the moduli space

The index is the largest integer dividing the canonical divisor in Pic. For now this is only implemented for the canonical stability condition.

EXAMPLES:

The usual 3-Kronecker example:

```
sage: from quiver import *
sage: Q = KroneckerQuiver(3)
```

(continues on next page)

(continued from previous page)

```
sage: QuiverModuliSpace(Q, (2, 3)).index()
3
```

Subspace quiver moduli have index 1:

```
sage: Q = SubspaceQuiver(7)
sage: QuiverModuliSpace(Q, (1, 1, 1, 1, 1, 1, 1, 2)).index()
1
```

chow_ring(*chi=None*, *classes=None*)

Returns the Chow ring of the moduli space.

For a given datum $(Q, \mathbf{d}, \theta)$ such that Q is acyclic and $\mathbf{d}$ is θ-coprime, the Chow ring of the moduli space of quiver representations is described in MR3318266 and arXiv.2307.01711.

Let

$$R = \bigotimes i \in Q_0 \mathbb{Q}[x_{i,1}, \ldots, x_{i,d_i}]$$

Let $e_{i,j}$ be the elementary symmetric function of degree j in d_i variables, and let $\xi_{i,j}$ be $e_{i,j}(x_{i,1}, \ldots, x_{i,d_i})$. We denote by A the ring of invariants

$$A := R^{S_\mathbf{d}} = \mathbb{Q}[\xi_{i,j}],$$

where $S_\mathbf{d} = \prod_{i \in Q_0} S_{\mathbf{d}_i}$ acts by permuting the variables.

The ring CH$(M^{\theta-st}(Q,d))$ is a quotient of *A* by two types of relations: a single linear relation, given by the choice of linearization upon which the universal bundles are constructed, and the so-called tautological relations, which we define below.

The *linear relation* given by the linearization *a* is the identity $\sum_{i \in Q_0} a_i c_1(U_i) = 0$ in A.

A subdimension vector $\mathbf{e}$ of *:math:{bf d}* is said to be "forbidden" if $\mu_\theta(\mathbf{e}) > \mu_\theta(\mathbf{d})$. One actually only needs to consider forbidden dimension vectors that are minimal with respect to a certain partial order, see *Quiver.division_order()*.

We define the *tautological ideal* I_{taut} of *R* as the ideal generated by the polynomials

$$\prod_{a \in Q_1} \prod_{k=1}^{e_{s(a)}} \prod_{\ell=d_{t(a)}+1}^{d_{t(a)}} \left(x_{t(a),\ell} - x_{s(a),k}\right),$$

for every forbidden subdimension vector *e* of *d*.

The tautological relations in *A* are then given by the image of I_{taut} under the *antisymmetrization* map

$$\rho : R \to A : \frac{1}{\delta} \sum_{\sigma \in S_\mathbf{d}} sign(\sigma)\sigma \cdot f,$$

where δ is the discriminant $\prod_{i \in Q_0} \prod_{1 \le k < \ell \le d_i} (x_{i,\ell} - x_{i,k})$.

The Chow ring CH$(M^{\theta-\text{st}}(Q,d))$ is then the quotient of *A* by $(\sum_{i \in Q_0} a_i c_1(U_i)) + \rho(I_{taut})$.

INPUT:

- `chi` – choice of linearization, we need that $\chi(\mathbf{d}) = 1$
- `classes` – list of generators for the polynomial ring (default: None)

OUTPUT: ring

EXAMPLES:

The Kronecker quiver:

```
sage: from quiver import *
sage: Q= KroneckerQuiver()
sage: X = QuiverModuliSpace(Q, (1, 1))
sage: chi = (1, 0)
sage: A = X.chow_ring(chi=chi)
sage: I = A.defining_ideal()
sage: [I.normal_basis(i) for i in range(X.dimension()+1)]
[[1], [x1_1]]
```

The 3-Kronecker quiver:

```
sage: from quiver import *
sage: Q = GeneralizedKroneckerQuiver(3)
sage: X = QuiverModuliSpace(Q, (2, 3))
sage: chi = (-1, 1)
sage: A = X.chow_ring(chi=chi)
sage: I = A.defining_ideal()
sage: [I.normal_basis(i) for i in range(X.dimension()+1)]
[[1],
[x1_1],
[x0_2, x1_1^2, x1_2],
[x1_1^3, x1_1*x1_2, x1_3],
[x1_1^2*x1_2, x1_2^2, x1_1*x1_3],
[x1_2*x1_3],
[x1_3^2]]
```

The 5-subspace quiver:

```
sage: from quiver import *
sage: Q, d = SubspaceQuiver(5), (1, 1, 1, 1, 1, 2)
sage: theta = (2, 2, 2, 2, 2, -5)
sage: X = QuiverModuliSpace(Q, d, theta, condition="semistable")
sage: chi = (-1, -1, -1, -1, -1, 3)
sage: A = X.chow_ring(chi=chi)
sage: I = A.defining_ideal()
sage: [I.normal_basis(i) for i in range(X.dimension()+1)]
[[1], [x1_1, x2_1, x3_1, x4_1, x5_1], [x5_2]]
```

The ideal Chow ring for our favourite 6-fold has 10 generators, 9 from the tautological ideal, and 1 linear relation:

```
sage: from quiver import *
sage: Q = GeneralizedKroneckerQuiver(3)
sage: X = QuiverModuliSpace(Q, (2, 3))
sage: chi = (-1, 1)
sage: R = X.chow_ring(chi=chi);
sage: R.ambient()
Multivariate Polynomial Ring in x0_1, x0_2, x1_1, x1_2, x1_3
over Rational Field
sage: len(R.defining_ideal().gens())
10
```

chern_class_line_bundle(*eta*, *classes=None*)

Returns the first Chern class of the line bundle

$$L(\eta) = \bigotimes_{i \in Q_0} \det(U_i)^{-\eta_i},$$

where η is a character of PG_d.

INPUT:

- `eta` – character of PG_d as vector in $\mathbb{Z}Q_0$

EXAMPLES:

On the Kronecker 6-fold we can take the canonical line bundle, which we can see to have index 3:

```
sage: from quiver import *
sage: Q = KroneckerQuiver(3)
sage: X = QuiverModuliSpace(Q, (2, 3))
sage: eta = Q.canonical_stability_parameter((2, 3))
sage: X.chern_class_line_bundle(eta)
-3*x1_1bar
```

chern_character_line_bundle(*eta*, *classes=None*)

Computes the Chern character of L(eta).

The Chern character of a line bundle L with first Chern class x is given by $e^x = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \ldots$

EXAMPLES:

On the Kronecker 6-fold the canonical line bundle has the following Chern character:

```
sage: from quiver import *
sage: Q = KroneckerQuiver(3)
sage: X = QuiverModuliSpace(Q, (2, 3))
sage: eta = Q.canonical_stability_parameter((2, 3))
sage: X.chern_character_line_bundle(eta)
4617/80*x1_3bar^2 - 1539/40*x1_2bar*x1_3bar + 81/8*x1_1bar^2*x1_2bar
- 27/8*x1_2bar^2 - 27/4*x1_1bar*x1_3bar - 9/2*x1_1bar^3 + 9/2*x1_1bar^2
- 3*x1_1bar + 1
```

total_chern_class_universal(*i*, *chi*, *classes=None*)

Gives the total Chern class of the universal bundle U_i(chi).

EXAMPLES:

The two summands for the Kronecker 6-fold:

```
sage: from quiver import *
sage: Q = KroneckerQuiver(3)
sage: X = QuiverModuliSpace(Q, (2, 3))
sage: chi = (-1, 1)
sage: X.total_chern_class_universal(0, chi)
x0_2bar + 2*x1_1bar + 1
sage: X.total_chern_class_universal(1, chi)
x0_2bar + x1_1bar + 1
```

point_class(*chi=None*, *classes=None*)

Returns the point class as an expression in Chern classes of the U_i (`chi`).

The point class is given as the homogeneous component of degree dim X of the expression

$$\prod_{a \in Q_1} c(U_{t(a)})^{d_{s(a)}} / (\prod_{i \in Q_0} c(U_i)^{d_i})$$

EXAMPLES

$\mathbb{P}^7$ as a quiver moduli space of a generalized Kronecker quiver:

```
sage: from quiver import *
sage: Q = GeneralizedKroneckerQuiver(8)
sage: X = QuiverModuliSpace(Q, (1, 1))
sage: chi = (1, 0)
sage: X.point_class(chi, classes=["o", "h"])
h^7
```

Our favorite 6-fold:

```
sage: from quiver import *
sage: Q = GeneralizedKroneckerQuiver(3)
sage: X = QuiverModuliSpace(Q, (2, 3))
sage: chi = (-1, 1)
sage: X.point_class(chi, classes=["x1", "x2", "y1", "y2", "y3"])
y3^2
```

A moduli space of the 5-subspace quiver; it agrees with the blow-up of $\mathbb{P}^2$ in 4 points in general position:

```
sage: from quiver import *
sage: Q = SubspaceQuiver(5)
sage: theta = (2, 2, 2, 2, 2, -5)
sage: X = QuiverModuliSpace(Q, (1, 1, 1, 1, 1, 2))
sage: chi = (-1, -1, -1, -1, -1, 3)
sage: X.point_class(chi, classes=['x1', 'x2', 'x3', 'x4', 'x5', 'y', 'z'])
1/2*z
```

If we don't specify *chi* a default that will still work is used, but the results do depend on it:

```
sage: X.point_class(classes=['x1', 'x2', 'x3', 'x4', 'x5', 'y', 'z'])
-1/3*y^2
```

degree (*eta=None*, *classes=None*)

Computes the degree of the line bundle given by eta.

INPUT:

- `eta` – class of line bundle (default: anticanonical line bundle
- `classes` – variables to be used (default: None)

EXAMPLES:

sage: from quiver import * sage: Q = KroneckerQuiver(3) sage: d = (2, 3) sage: X = QuiverModuliSpace(Q, d) sage: eta = Q.canonical_stability_parameter(d) sage: eta = eta / 3 sage: X.degree(eta) 57

todd_class()

The Todd class of X is the Todd class of the tangent bundle.

Currently not yet implemented.

$$td(X) = (\prod_{a:i\to j\in Q_1}\prod_{p=1}^{d_j}\prod_{q=1}^{d_i} Q(t_{j,q}-t_{i,p}))/(prod_{i\in Q_0}\prod_{p,q=1}^{d_i} Q(t_{i,q}-t_{i,p}))$$

EXAMPLES:

Not yet implemented:

```
sage: from quiver import *
sage: Q = KroneckerQuiver(3)
sage: X = QuiverModuliSpace(Q, (2, 3))
sage: X.todd_class()
Traceback (most recent call last):
...
NotImplementedError
```

class quiver.**QuiverModuliStack**(*Q*, *d*, *theta=None*, *denom=<built-in function sum>*, *condition='semistable'*)

__init__(*Q*, *d*, *theta=None*, *denom=<built-in function sum>*, *condition='semistable'*)

Constructor for a quiver moduli stack.

This is the quiver moduli space as a stack.

INPUT:

- `Q` – quiver
- `d` — dimension vector
- `theta` – stability parameter (default: canonical stability parameter)
- `denom` – denominator for slope stability (default: `sum`), needs to be effective on the simple roots
- `condition` – whether to include all semistables, or only stables (default: "semistable")

EXAMPLES:

An example:

```
sage: from quiver import *
sage: Q = KroneckerQuiver(3)
sage: X = QuiverModuliStack(Q, (2, 3))
```

repr()

Give a shorthand string presentation for a quiver moduli stack.

EXAMPLES:

A Kronecker moduli spac:

```
sage: from quiver import *
sage: Q = KroneckerQuiver(3)
sage: QuiverModuliStack(Q, (2, 3))
moduli stack of semistable representations, with
- Q = 3-Kronecker quiver
- d = (2, 3)
- θ = (9, -6)
```

dimension()

Computes the dimension of the moduli stack $[R^{(s)st}/G]$.

This is the dimension of a quotient stack, thus we use

$$dim[R^{(s)st}/G] = dimR^{(s)st} - dimG$$

The dimension turns out to be $-\langle d, d\rangle$ if the (semi-)stable locus is non-empty.

EXAMPLES:

The dimension of a moduli space of stable is off by one from the moduli stack because of the generic stabilizer being 1-dimensional:

```
sage: from quiver import *
sage: Q = KroneckerQuiver(3)
sage: X = QuiverModuli(Q, (2, 3))
sage: X.to_stack().dimension()
5
sage: X.to_space().dimension()
6
```

is_smooth() → bool

Return whether the stack is smooth.

The stack is a quotient of a smooth variety, thus it is always smooth.

EXAMPLES:

Nothing interesting to see here:

```
sage: from quiver import *
sage: QuiverModuliSpace(KroneckerQuiver(3), (2, 3)).is_smooth()
True
```

motive()

Gives an expression for the motive of the semistable moduli stack

This really lives inside an appropriate localization of K_0(Var), but it only involves the Lefschetz class.

EXAMPLES:

Loop quivers:

```
sage: from quiver import *
sage: Q = LoopQuiver(0)
sage: X = QuiverModuliStack(Q, (2,), (0,))
sage: X.motive()
1/(L^4 - L^3 - L^2 + L)
sage: Q = LoopQuiver(1)
sage: X = QuiverModuliStack(Q, (2,), (0,))
sage: X.motive()
L^3/(L^3 - L^2 - L + 1)
```

The 3-Kronecker quiver:

```
sage: Q = GeneralizedKroneckerQuiver(3)
sage: X = QuiverModuliStack(Q, (2, 3))
sage: X.motive()
(-L^6 - L^5 - 3*L^4 - 3*L^3 - 3*L^2 - L - 1)/(L - 1)
```

chow_ring(*classes=None*)

Returns the Chow ring of the quotient stack.

INPUT:

- `classes` – variables to be used (default: None)

EXAMPLES:

The Chow ring of the stack defining the Kronecker 6-fold has as its defining ideal the tautological ideal:

```
sage: from quiver import *
sage: Q = KroneckerQuiver(3)
sage: X = QuiverModuliStack(Q, (2, 3))
sage: X.tautological_ideal() == X.chow_ring().defining_ideal()
True
```

CHAPTER

THREE

CONSTRUCTING QUIVERS

quiver.**disjoint_union**(*Q1*, *Q2*)

Returns the disjoint union of two quivers Q1 and Q2.

EXAMPLES:

We construct the disjoint union of 2 generalized Kronecker quivers:

```
sage: from quiver import *
sage: Q1 = GeneralizedKroneckerQuiver(3)
sage: Q2 = GeneralizedKroneckerQuiver(4)
sage: Q = disjoint_union(Q1,Q2)
sage: Q
disjoint union of 3-Kronecker quiver and 4-Kronecker quiver
```

quiver.**GeneralizedKroneckerQuiver**(*m: int*)

Return the generalized Kronecker quiver

The generalized Kronecker quiver has two vertices and m arrows from the first to the second vertex.

INPUT:

- m – integer; number of arrows in the quiver

OUTPUT: the generalized Kronecker quiver as Quiver instance

EXAMPLES:

The generalized Kronecker quiver is as claimed:

```
sage: from quiver import *
sage: Q = GeneralizedKroneckerQuiver(3)
sage: Q.number_of_vertices()
2
sage: Q.number_of_arrows()
3
```

quiver.**KroneckerQuiver**(*m: int = 2*)

Return the Kronecker quiver

The Kronecker quiver has two vertices and 2 arrow from the first to the second vertex. If the optional parameter m is specified we construct the generalized Kronecker quiver on m arrows;

INPUT:

- m – integer (default: *2*); number of arrows in the quiver

OUTPUT: the Kronecker quiver as Quiver instance

EXAMPLES:

The Kronecker quiver is as claimed:

```
sage: from quiver import *
sage: Q = KroneckerQuiver()
sage: Q.number_of_vertices()
2
sage: Q.number_of_arrows()
2
```

If we specify the number of arrows, we construct a generalized Kronecker quiver:

```
sage: KroneckerQuiver(3) == GeneralizedKroneckerQuiver(3)
True
```

quiver.**ThreeVertexQuiver**(*m12: int*, *m13: int*, *m23: int*)

Constructs a 3-vertex quiver, with and $m_{i,j}$ arrows from *i* to *j*.

Thus it is always an acyclic quiver.

INPUT:

- m12 – integer; number of arrows from 1 to 2
- m13 – integer; number of arrows from 1 to 3
- m23 – integer; number of arrows from 2 to 3

EXAMPLES:

A 3-vertex quiver with 5 arrows:

```
sage: from quiver import *
sage: Q = ThreeVertexQuiver(2, 2, 1); Q
an acyclic 3-vertex quiver of type (2, 2, 1)
sage: Q.number_of_arrows()
5
```

quiver.**LoopQuiver**(*m: int*)

Return the quiver with 1 vertex and m loops.

This is a synonym for *GeneralizedJordanQuiver()*.

EXAMPLES:

This is a synonym:

```
sage: from quiver import *
sage: LoopQuiver(7) == GeneralizedJordanQuiver(7)
True
```

See also:

GeneralizedJordanQuiver()

EXAMPLES:

The loop quiver with 7 loops:

```
sage: from quiver import *
sage: Q = LoopQuiver(7)
sage: Q.adjacency_matrix()
[7]
```

quiver.**JordanQuiver**(*m: int = 1*)

Return the generalized Jordan quiver with m loops

The default value for m is 1.

This is a synonym:

```
sage: from quiver import *
sage: JordanQuiver(7) == GeneralizedJordanQuiver(7)
True
```

See also:

GeneralizedJordanQuiver()

EXAMPLES:

The Jordan quiver has one loop:

```
sage: from quiver import *
sage: Q = JordanQuiver()
sage: Q.adjacency_matrix()
[1]
```

quiver.**GeneralizedJordanQuiver**(*m: int*)

Return the generalized Jordan quiver with m loops

INPUT:

- m – integer; the number of loops in the generalized Jordan quiver

OUTPUT: the generalized Jordan quiver

EXAMPLES:

The generalized Jordan quiver has 1 vertex and m arrows:

```
sage: from quiver import *
sage: Q = GeneralizedJordanQuiver(7)
sage: Q.number_of_vertices()
1
sage: Q.number_of_arrows()
7
```

quiver.**SubspaceQuiver**(*m: int*)

Return the subspace quiver with m sources

The sources are labelled $1, \ldots, m$ and the sink is $m+1$; there is one arrow from every source to the sink.

INPUT:

- m – integer; the number of sources in the subspace quiver

OUTPUT: the subspace quiver with m sources

EXAMPLES:

The subspace quiver with m sources has m arrows and m+1 vertices:

```
sage: from quiver import *
sage: Q = SubspaceQuiver(6)
sage: Q.number_of_vertices()
7
sage: Q.number_of_arrows()
6
```

The subspace quiver with 2 sources is also a 3-vertex quiver:

```
sage: SubspaceQuiver(2) == ThreeVertexQuiver(0, 1, 1)
True
```

quiver.**ThickenedSubspaceQuiver**(*m*, *k*)

Return the thickened subspace quiver with m sources

The sources are labelled $1, \ldots, m$ and the sink is $m+1$; there are k arrows from every source to the sink.

- m – integer; the number of sources in the subspace quiver
- k – integer; the number arrows from a source to the sink

OUTPUT: the subspace quiver with m sources and k arrows from each source

EXAMPLES:

The k-thickened subspace quiver with m sources has km arrows, m+1 vertices:

```
sage: from quiver import *
sage: Q = ThickenedSubspaceQuiver(6, 2)
sage: Q.number_of_vertices()
7
sage: Q.number_of_arrows()
12
```

The k-thickened subspace quiver with 2 sources is also a 3-vertex quiver:

```
sage: ThickenedSubspaceQuiver(2, 6) == ThreeVertexQuiver(0, 6, 6)
True
```

quiver.**GeneralizedSubspaceQuiver**(*m*, *K*)

Return the generalized subspace quiver with m sources and multiplicities K

The sources are labelled $1, \ldots, m$ and the sink is $m+1$; there are are K_i arrows from every source $i = 1, \ldots, m$ to the sink.

- m – integer; the number of sources in the subspace quiver
- K – list of integers; the number arrows from the *i*-th source to the sink

OUTPUT: the subspace quiver with m sources and K_i arrows from each source

EXAMPLES:

The generalized subspace quiver with *m* sources and multiplicities *K* has $\sum_{i=1}^{m} K_i$ arrows and $m+1$ vertices:

```
sage: from quiver import *
sage: Q = GeneralizedSubspaceQuiver(6, (1, 2, 3, 4, 5, 6))
sage: Q.number_of_vertices()
7
sage: Q.number_of_arrows()
21
```

The generalized subspace quiver with 2 sources is also a 3-vertex quiver:

```
sage: GeneralizedSubspaceQuiver(2, (2, 3)) == ThreeVertexQuiver(0, 2, 3)
True
```

quiver.**DynkinQuiver**(*T*)

Return the Dynkin quiver of type *T*

The type *T* is to be taken as in the Sage method *DynkinDiagram*, and the quiver is oriented lexigraphically in the vertices of the diagram.

INPUT:

- T: a Dynkin type, as documented in the Sage method *DynkinDiagram*

OUTPUT: the Dynkin quiver with lexicographic ordering on the vertices

EXAMPLES:

The A_2 quiver is the generalized Kronecker quiver with 1 arrow:

```
sage: from quiver import *
sage: DynkinQuiver("A2") == GeneralizedKroneckerQuiver(1)
True
```

The Dynkin quiver D_4 is a different orientation of the 3-subspace quiver:

```
sage: DynkinQuiver("D4") == SubspaceQuiver(3)
False
```

We can also consider disconnected Dynkin quivers:

```
sage: Q = DynkinQuiver("A3xA4")
sage: Q.is_connected()
False
```

quiver.**ExtendedDynkinQuiver**(*T*)

Return the Dynkin quiver of type *T*

The type *T* is a string which can be passed onto the Sage method *DynkinDiagram*, and the quiver is oriented lexigraphically in the vertices of the diagram. The special vertex thus comes first.

INPUT:

- T – string: a Dynkin type, as documented in the Sage method *DynkinDiagram*

OUTPUT: the extended Dynkin quiver with lexicographic ordering on the vertices

EXAMPLES:

The extended A_1 quiver is the Kronecker quiver:

```
sage: from quiver import *
sage: ExtendedDynkinQuiver("A1") == KroneckerQuiver()
True
```

quiver.**CyclicQuiver**(*n*)

Return the cyclic quiver on n vertices

This is the quiver with n vertices and n arrows from *i* to *i+1* for $i = 1, \ldots, n$, with *n+1=1*.

INPUT:

- n– integer; the number of vertices (and arrows)

OUTPUT: cyclic quiver on n vertices

EXAMPLES:

The doubled Dynkin quiver of type A_2 is also the cyclic quiver on 2 vertices:

```
sage: from quiver import *
sage: CyclicQuiver(2) == DynkinQuiver("A2").doubled_quiver()
True
```

quiver.**BipartiteQuiver**(*m*, *n*)

Return the bipartite quiver with m sources and n sinks

This is the quiver with $m+n$ vertices, having 1 arrow from each of the first m vertices to the each of the last n vertices.

INPUT:

- m – non-negative integer; number of sources
- n – non-negative integer; number of sinks

OUTPUT: bipartite quiver with m sources and n sinks

EXAMPLES:

When *m=n=1* we get the A_2 quiver:

```
sage: from quiver import *
sage: BipartiteQuiver(1, 1) == DynkinQuiver("A2")
True
```

When *m=2* and *n=1* we get a 3-vertex quiver:

```
sage: BipartiteQuiver(2, 1) == ThreeVertexQuiver(0, 1, 1)
True
```

INDEX

Non-alphabetical

A

B

C

D

E

F

G

T

V

Z