# QuiverTools

***Release v0***

**unknown**

**Jul 02, 2024**

# CONTENTS

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

**__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:

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

A Dynkin quiver of type A_3:

```
sage: Q = Quiver.from_string("1-2-3"); Q.adjacency_matrix()
[0 1 0]
[0 0 1]
[0 0 0]
```

A triangle-shaped quiver:

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

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

Construct a quiver from a DiGraph object.

INPUT:

- `G` – directed graph
- `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.

OUTPUT: the quiver

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

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

Return str(self).

**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
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()

sage: Q.rename("3-Kronecker quiver")
sage: Q.get_custom_name()
'3-Kronecker quiver'
sage: Q.reset_name()
sage: Q.get_custom_name()

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
```

**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
```

(continues on next page)

(continued from previous page)

```
sage: Q.is_source("b")
False
```

**is_sink**(*i*) → 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**($x$, $y$) $\to$ 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**($x$) $\to$ 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
```

(continues on next page)

(continued from previous page)

```
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 *iin 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**

File: /__w/QuiverTools/QuiverTools/quiver/quiver.py (starting at line 1300)

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**

File: /__w/QuiverTools/QuiverTools/quiver/quiver.py (starting at line 1331)

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**

File: /__w/QuiverTools/QuiverTools/quiver/quiver.py (starting at line 1362)

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**$(x) \to$ 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
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())
```

(continues on next page)

(continued from previous page)

```
False
sage: Q.is_imaginary_root([4, 1])
False
```

**is_schur_root**(*d*) → 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 *d* 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),
```

(continues on next page)

(continued from previous page)

```
     (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)
[]
```

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 $\theta$-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)
```

(continues on next page)

(continued from previous page)

```
True
sage: Q.is_theta_coprime([3, 3], [1, -1])
False
```

**is_indivisible**($d$) $\to$ 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 = vector([1, 1, 1])
sage: Q.support(d)
[0, 1, 2]
sage: d = vector([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

- $\mathrm{supp}(\mathbf{d})$ 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])
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 *dll 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._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**

File: /__w/QuiverTools/QuiverTools/quiver/quiver.py (starting at line 2027) 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 a result of Schofield (see Thm. 5.3 of https://arxiv.org/pdf/0802.2147.pdf) 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
```

(continues on next page)

(continued from previous page)

```
(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
(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
```

(continues on next page)

(continued from previous page)

```
(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 = vector([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)
[(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)]
```

(continues on next page)

(continued from previous page)

```
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 $\mathrm{ext}(d, e)$.

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

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

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
```

**all_hn_types**

File: /__w/QuiverTools/QuiverTools/quiver/quiver.py (starting at line 2350) Returns the list of all Harder–Narasimhan types of d.

INPUT:

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

OUTPUT: list of Harder–Narasimhan types

EXAMPLES:

The Harder–Narasimhan types for the 3-Kronecker quiver:

```
sage: from quiver import *
sage: Q = GeneralizedKroneckerQuiver(3)
sage: d = (2, 3)
sage: theta = (3, -2)
sage: Q.all_hn_types(d, theta)
[((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),)]
```

**canonical_stability_parameter**(*d*)

Returns the canonical stability parameter for d

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}
```

**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 **MR1162486_** a dimension vector *d* admits a $\theta$-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
```

(continues on next page)

(continued from previous page)

```
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 representation 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 $\mathrm{A}_2$ 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**

File: /__w/QuiverTools/QuiverTools/quiver/quiver.py (starting at line 2578)

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 $i \neq j, \mathrm{ext}(e_i, e_j) = \mathrm{ext}(e_j, e_i) = 0$.

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)]
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.

EXAMPLES:

The first Hochschild cohomology of the $m$-th generalized Kronecker quiver is the dimension of $\mathrm{PGL}_{m+1}$:

```
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
```

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

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

Returns the list of all HN types.

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

- $d^1 + ... + d^s = d$
- $\mu_\theta(d^1) > ... > \mu_\theta(d^s)$
- Every $d^k$ is $\theta$-semi-stable.

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: d = [2, 3, 2]
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)),
```

(continues on next page)

(continued from previous page)

```
((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)),
((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 HN type.

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

- $d^1 + ... + d^s = d$
- $\mu_\theta(d^1) > ... > \mu_\theta(d^s)$
- Every $d^k$ is theta-semi-stable.

INPUT:

- `dstar` – list of vectors of Ints

OUTPUT: statement truth value as Bool

EXAMPLES:

The 3-Kronecker quiver:

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

INPUT:

- `dstar` – list of vectors of Ints
- `secure` – (default: False): Bool

OUTPUT: codimension as Int # TODO # It checks for dstar to be a HN type iff secure == True. This check is slow. # Be sure to be dealing with a HN type if you call it with secure == False. This is fast but yields nonsense, if dstar is not a HN type.

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

$$-sum_{k<l} < d^k, d^l >$$

EXAMPLES

The 3-Kronecker quiver:

```
sage: from quiver import *
sage: Q, d = GeneralizedKroneckerQuiver(3), vector([2,3])
sage: theta = vector([1,0])
sage: X = QuiverModuliSpace(Q, d, theta)
sage: hn = X.all_harder_narasimhan_types(); hn
[((1, 0), (1, 1), (0, 2)),
```

(continues on next page)

(continued from previous page)

```
 ((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.codimension_of_harder_narasimhan_stratum(dstar) for dstar in hn]
[12, 9, 8, 3, 18, 10, 4, 0]
```

**codimension_unstable_locus**()

Computes the codimension of the unstable locus inside the representation variety.

OUTPUT: codimension as Int

EXAMPLES:

The 3-Kronecker quiver:

```
sage: from quiver import *
sage: Q = GeneralizedKroneckerQuiver(3)
sage: X = QuiverModuliSpace(Q, [2, 3], [1, 0])
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 Kronecker quiver:

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

**all_luna_types**()

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

OUTPUT: list of tuples containing Int-vector and Int

A Luna type of d for theta is an unordered sequence (i.e. multiset) $((d^1, m_1), ..., (d^s, m_s))$ of dimension vectors $d^k$ and positive integers $m_k$ such that

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

Example: Suppose that *d = 3e* and *e, 2e, d = 3e* are the only stable subdimension vectors. Then the Luna

types are

$$((3e,1))$$
$$((2e,1),(e,1))$$
$$((e,3))$$
$$((e,2),(e,1))$$
$$((e,1),(e,1),(e,1)).$$

We implement it as follows.

A Luna type for us is a dictionary `{d^1: p_1^1,..., d^s: p_s^1,..., d_s: p_s^t}` of dimension vectors d^k and non-empty partitions p^k such that

$$|p_1^1|d^1 + ... + |p_s^t|d^s = d$$

So in the above example, the Luna types are:

```
{3e: [1]}
{2e: [1], e: [1]}
{e: [3]}
{e: [2, 1]}
{e: [1, 1, 1]}
```

EXAMPLES:

The Kronecker quiver:

```
sage: from quiver import *
sage: Q, d, theta = KroneckerQuiver(), vector([3,3]), vector([1,-1])
sage: X = QuiverModuliSpace(Q, d, theta)
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, d = GeneralizedKroneckerQuiver(3), vector([3,3])
sage: theta = vector([1,-1])
sage: X = QuiverModuliSpace(Q, d, theta)
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, d, theta = KroneckerQuiver(), vector([0,0]), vector([1,-1])
sage: X = QuiverModuliSpace(Q, d, theta)
sage: X.all_luna_types()
[{(0, 0): [1]}]
```

**is_luna_type**(*tau*) → bool

Checks if tau is a Luna type for theta.

INPUT:

- `tau` – dictionary with dimension vectors as keys and lists of ints as values

OUTPUT: whether `tau` is a Luna type.

EXAMPLES:

The Kronecker quiver:

```
sage: from quiver import *
sage: Q, d, theta = KroneckerQuiver(), vector([3,3]), vector([1,-1])
sage: X = QuiverModuliSpace(Q, d, theta)
sage: l = X.all_luna_types()
sage: all(X.is_luna_type(tau) for tau in l)
True
```

The 3-Kronecker quiver with zero vector:

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

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

Computes the dimension of the Luna stratum $S_\tau$.

INPUT:

- `tau` – list of tuples
- `secure` – Bool

OUTPUT: Dimension as Int

The dimension of the Luna stratum of `tau = {d^1: p^1,...,d^s: p^s}` is $\sum_k l(p^k)(1- < \langle d^k, d^k \rangle)$, 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, d, theta = KroneckerQuiver(), vector([2,2]), vector([1,-1])
sage: X = QuiverModuliSpace(Q, d, theta)
sage: L = X.all_luna_types(); L
[{(1, 1): [2]}, {(1, 1): [1, 1]}]
sage: [X.dimension_of_luna_stratum(tau) for tau in L]
[1, 2]
```

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

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

INPUT:

- `tau` – list of tuples

- secure – Bool

OUTPUT: tuple consisting of a Quiver object and a vector

EXAMPLES:

The 3-Kronecker quiver:

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

**codimension_properly_semistable_locus**()

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

OUTPUT: codimension as Int

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

**semistable_equals_stable**()

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

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

OUTPUT: whether every theta-semistable representation is theta-stable for `self._theta`

EXAMPLES:

The 3-Kronecker quiver:

```
sage: from quiver import *
sage: Q, d = GeneralizedKroneckerQuiver(3), vector([3,3])
sage: theta = vector([1,-1])
sage: X = QuiverModuliSpace(Q, d, theta)
sage: X.semistable_equals_stable()
False
sage: e = vector([2,3])
sage: Y = QuiverModuliSpace(Q, e, theta)
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: Q.is_theta_coprime(d, theta)
False
sage: X = QuiverModuliSpace(Q, d, theta)
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 $\theta$-amply stable if the codimension of the $\theta$-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 $d$ strongly amply stable for $\theta$ if $\langle e, d-e\rangle \leq -2$ holds for all subdimension vectors $e$ of $d$ which satisfy $\mu_\theta(e) >= \mu_\theta(d)$.

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)
```

(continues on next page)

(continued from previous page)

```
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

**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 $\det(N_{S/R}|_Z)$ with a certain weight. Teleman quantization gives a numerical condition involving these weights to compute cohmology 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:

sage: from quiver import * sage: X = QuiverModuliSpace(KroneckerQuiver(3), [2, 3]) sage: X.if_rigidity_inequality_holds() True sage: X = QuiverModuliSpace(ThreeVertexQuiver(1, 6, 1), [1, 6, 6]) sage: X.if_rigidity_inequality_holds() False

**all_minimal_forbidden_subdimension_vectors**()

Returns the list of all *minimal* forbidden subdimension vectors

Minimality is with respect to the partial order *e* << *d* which means $e_i \leq d_i$ for every source *i*, $e_j \geq d_j$ for every sink *j*, and $e_k = d_k$ for every vertex which is neither a source nor a sink. See also *Quiver.division_order()*.

OUTPUT: list of minimal forbidden dimension vectors

EXAMPLES:

The 3-Kronecker quiver:

```
sage: from quiver import *
sage: Q = GeneralizedKroneckerQuiver(3)
sage: X = QuiverModuliSpace(Q, [3, 3], [1, -1], condition="semistable")
sage: X.all_minimal_forbidden_subdimension_vectors()
[(1, 0), (2, 1), (3, 2)]
sage: Y = QuiverModuliSpace(Q, [3, 3], [1, -1], condition="stable")
sage: Y.all_minimal_forbidden_subdimension_vectors()
[(1, 1), (2, 2)]
```

**tautological_relations**(*inRoots=False*, *chernClasses=None*, *chernRoots=None*)

Returns the tautological relations in Chern classes (if inRoots == False) or in Chern roots.

INPUT:

- `inRoots` – Bool
- `chernClasses` – list of Strings
- `chernRoots` – list of Strings

OUTPUT: list

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

**dimension**()

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

This involves several cases:

- If there are $\theta$-stable representations then $\dim M^{\theta-sst}(Q,d) = M^{\theta-st}(Q,d) = 1 - \langle d,d\rangle$;
- if there are no $\theta$-stable representations then $\dim M^{\theta-st}(Q,d) = -\infty$ by convention, and we define $\dim M^{\theta-sst} = \max_\tau\{\dim S_\tau\}$, 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)
```

(continues on next page)

(continued from previous page)

```
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")
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
```

**poincare_polynomial()**

Returns the Poincare polynomial of the moduli space.

OUTPUT: polynomial in one variable # TODO allow a user-supplied ring?

The Poincare polynomial is defined as

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

For a quiver moduli space whose dimension vector is $\theta$-coprime, the odd cohomology vanishes and this is a Polynomial in $q$. We use Cor. 6.9 in Reineke's Harder–Narasimhan paper to compute it.

EXAMPLES:

Some Kronecker quivers:

```
sage: from quiver import *
sage: Q = KroneckerQuiver()
sage: X = QuiverModuliSpace(Q, [1, 1])
sage: X.poincare_polynomial()
q + 1
```

(continues on next page)

(continued from previous page)

```
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: List of Ints

EXAMPLES:

Some Kronecker quivers:

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

**picard_rank**()

Computes the Picard rank of the moduli space for known cases.

**index**()

Computes the index of the moduli space for known cases, i.e., the largest integer dividing the canonical divisor in Pic.

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

Returns the Chow ring of the moduli space.

INPUT:

- `chi` – vector of Ints
- `chernClasses` – list of Strings

OUTPUT: ring

EXAMPLES:

The Kronecker quiver:

```
sage: from quiver import *
sage: Q, d, theta = KroneckerQuiver(), vector([1,1]), vector([1,-1])
sage: X = QuiverModuliSpace(Q, d, theta, condition="semistable")
sage: chi = vector([1,0])
```

(continues on next page)

(continued from previous page)

```
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, d = GeneralizedKroneckerQuiver(3), vector([2,3])
sage: theta = vector([3,-2])
sage: X = QuiverModuliSpace(Q, d, theta, condition="semistable")
sage: chi = vector([-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-subspaces quiver:

```
sage: from quiver import *
sage: Q, d = SubspaceQuiver(5), vector([1,1,1,1,1,2])
sage: theta = vector([2,2,2,2,2,-5])
sage: X = QuiverModuliSpace(Q, d, theta, condition="semistable")
sage: chi = vector([-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]]
```

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

Returns the first Chern class of the line bundle $L(\eta) = \bigotimes_{i \in Q_0} \det(U_i)^{-\eta_i}$, where $\eta$ is a character of $PG_d$.

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

Computes the Chern character of L(eta).

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

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

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

**point_class**(*chi=None*, *chernClasses=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: d = vector([1,1])
sage: theta = vector([1,-1])
sage: X = QuiverModuliSpace(Q,d,theta,condition="semistable")
sage: chi = vector([1,0])
sage: X.point_class(chi,chernClasses=['o','h'])
h^7
```

Our favorite 6-fold:

```
sage: from quiver import *
sage: Q = GeneralizedKroneckerQuiver(3)
sage: d = vector([2,3])
sage: theta = vector([3,-2])
sage: X = QuiverModuliSpace(Q,d,theta,condition="semistable")
sage: chi = vector([-1,1])
sage: X.point_class(chi,chernClasses=['x1','x2','y1','y2','y3'])
y3^2
```

A moduli space of the 5-subspaces 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: d = vector([1,1,1,1,1,2])
sage: theta = vector([2,2,2,2,2,-5])
sage: X = QuiverModuliSpace(Q,d,theta,condition="semistable")
sage: chi = vector([-1,-1,-1,-1,-1,3])
sage: X.point_class(chi,chernClasses=['x1','x2','x3','x4','x5','y','z'])
1/2*z
```

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

Computes the degree of the ample line bundle given by eta.

**todd_class**()

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

For quiver moduli it computes as

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

**class** quiver.**QuiverModuliStack**(*Q*, *d*, *theta*, *denom=<built-in function sum>*, *condition='semistable'*)

**dimension**()

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

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

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

**motive**()

Gives an expression for the motive of the semistable moduli stack in an appropriate localization of K_0(Var)

# TODO more explanation

EXAMPLES:

Loop quivers:

```
sage: from quiver import *
sage: Q, d, theta = LoopQuiver(0), vector([2]), vector([0])
sage: X = QuiverModuliStack(Q, d, theta, condition="semistable")
sage: X.motive()
1/(L^4 - L^3 - L^2 + L)
sage: Q, d, theta = LoopQuiver(1), vector([2]), vector([0])
sage: X = QuiverModuliStack(Q, d, theta, condition="semistable")
sage: X.motive()
L^3/(L^3 - L^2 - L + 1)
```

The 3-Kronecker quiver:

```
sage: Q, d = GeneralizedKroneckerQuiver(3), vector([2,3])
sage: theta = vector([3,-2])
sage: X = QuiverModuliStack(Q, d, theta, condition="semistable")
sage: X.motive()
(-L^6 - L^5 - 3*L^4 - 3*L^3 - 3*L^2 - L - 1)/(L - 1)
```

**chow_ring**(*chernClasses=None*)

Returns the Chow ring of the quotient stack.

INPUT:

- `chernClasses`: list of Strings

OUTPUT: ring

# PYTHON MODULE INDEX

## q

# INDEX

## Symbols



## A



## B



## C



## D



## E



## F



## G



## H

## I



## L



## M



## N



## O



## P



## Q



## R



## S



## T



## V



## Z