arXiv:1102.4844v2 [math.CO] 29 Mar 2011

A COMPENDIUM ON THE CLUSTER ALGEBRA AND QUIVER PACKAGE IN `sage`

GREGG MUSIKER AND CHRISTIAN STUMP

Contents

0. Preface 1
1. Introduction 3
2. What is a cluster algebra? 6
3. Using quivers as cluster algebra seeds 14
4. Finite type and finite mutation type classifications 21
4.1. Finite mutation type classification 26
4.2. Skew-symmetrizable cluster algebra seeds of finite mutation type 26
4.3. Class sizes of finite and affine quiver mutation types 30
4.4. Algorithms for computing mutation classes 32
5. Associahedra and the cluster complex 35
5.1. Generalized associahedra 36
5.2. The cluster complex 39
6. Methods and attributes 39
6.1. Skew-symmetrizable matrices 40
6.2. QuiverMutationType 40
6.3. Quiver 48
6.4. ClusterSeed 53
6.5. ClusterVariable 59
References 59

0. Preface

The idea for a cluster algebra and quiver package in the open-source computer algebra system `sage` was born during the `sage` days 20.5 which were held at the Fields Institute in May 2010. The purpose of this package is to provide a platform to work with cluster algebras in graduate courses and to further develop the theory by working on examples, by gathering data, and by exhibiting and testing conjectures. In this compendium, we include the relevant theory to introduce the reader to cluster algebras assuming no prior background; this exposition has been written to be accessible to an interested undergraduate.

Date: March 30, 2011.
Key words and phrases. cluster algebra, quiver, associahedra, `sage`, `sage-combinat`.

Version 1.0 of the software package and this compendium is the result of many discussions on mathematical background and on implementation algorithms, and of many, many hours of coding. It is part of the sage-combinat project [SageComb].

For more information on sage, in particular on a detailed description how to install the program, we refer the reader to http://www.sagemath.org [Sage]; for more on the sage-combinat project, see http://wiki.sagemath.org/combinat. Throughout this compendium, we include examples that the user can run in the sage-notebook or sage command line. The package provides as well an interactive mode for the sage-notebook as shown in Figure 1 at the end of Section 3. We will close with a detailed description of the data structures and methods in this package. We follow the usual sage convention of indexing all lists starting at zero.

Currently, installing the sage-combinat queue is a necessary requirement for working with the cluster algebra and quiver package. In order to install the sage-combinat queue, you have to, after installing sage, run the command

```
# ./sage -combinat install
```

on the unix command line. Once the sage-combinat branch is created, one can use the command

```
# ./sage -combinat update
```

to update to the latest version of the sage-combinat queue, or one can use the command

```
# ./sage -combinat upgrade
```

to update to the latest version of the sage-combinat queue and to upgrade sage to its newest version. For more detailed explanations, please visit the sage-combinat wiki page. Installing the sage-combinat queue will eventually become obsolete after the project has gone through testing and reviewing processes, which might take time due to the involvedness of the algorithms (especially on mutation type detections).

This current version 1.0 should not be considered a complete, unchangeable, totally stable version. We will keep working on this project by fixing bugs, improving algorithms, and by adding functionalities. So it might be a good idea to update the sage-combinat queue once in a while, especially if you have encountered a problem.

We anticipate this ongoing project being improved with feedback from users. We are very interested in getting any type of feedback: on ways in which the package has been used, on features people like or that could be done better, or requests for new functionalities. If you are interested in helping us make improvements or further develop this package, we would be happy to have you involved.

Several other people have also worked on software for computations involving cluster algebras and quiver representations. Links to these are available at Fomin's Cluster Algebra Portal http://www.math.lsa.umich.edu/~fomin/cluster.html. This software includes work of Chapoton [Cha], Dupont-Pérotin [DP], Keller [Kel], and L. Williams.

Acknowledgements. We thank Franco Saliola and Sébastien Labbé for help on details of the `sage` development process. We also thank Florian Block, Hugh Thomas, and Leandro Vendramin for several contributions in the early stages of this project. We thank William Stein, Florent Hivert, Nicolas Thiéry, and all of the developers of `sage` and `sage-combinat` for their continued work on this open-source mathematical software. Finally, we like to thank the Fields Institute for its hospitality during the `sage` days 20.5 in May 2010 where this project was initiated. We also thank Bernhard Keller for a careful reading and numerous helpful edits to this guide.

1. Introduction

Cluster algebras, invented by Fomin and Zelevinsky [FZ02a], are certain commutative algebras which are isomorphic to subalgebras of the field of rational functions. Each cluster algebra has a distinguished set of generators called cluster variables; this set is a union of overlapping algebraically independent finite subsets called clusters, which together have the structure of a simplicial complex. The clusters are related to each other by binomial exchange relations. In the past ten years, such algebras have been found to be related to a number of other topics such as quiver representations, tropical geometry, canonical bases of semisimple algebraic groups, total positivity, generalized associahedra, Poisson geometry, and Teichmüller theory.

Usually, when one defines an *algebra* $\mathcal{A}$, one describes it by writing down the *generators* and *relations* of $\mathcal{A}$. Instead, when working with a *cluster algebra*, only a finite set of generators are provided at first, along with combinatorial data that allows one to algebraically construct the rest of the generators by applying a sequence of exchange rules. With this definition in mind, a *seed* for a cluster algebra $\mathcal{A}$ is a pair $(\mathbf{x}, B)$, where $\mathbf{x}$ denotes the *initial cluster*, and B denotes an *exchange matrix* (or B-matrix)[1]. Here, the cluster $\mathbf{x}$ consists of exchangeable generators, known as *cluster variables* and non-exchangeable generators, known as *coefficients* or *frozen variables*.

One of the simplest families of cluster algebras are those which are coefficient-free and rank two. Such algebras are parametrized by two positive integers (b, c), and the associated cluster algebra $\mathcal{A}(b, c)$ is defined to be the algebra generated by the set $\{x_n\}_{n\in\mathbb{Z}}$, where for $n \notin \{0, 1\}$,

$$x_n = \frac{x_{n-1}^b + 1}{x_{n-2}} \text{ if } n \text{ is even, and } \frac{x_{n-1}^c + 1}{x_{n-2}} \text{ if } n \text{ is odd.}$$

These are implemented in `sage`, for example (letting $b = 2$, and $c = 3$) as

```
sage:  S23 = ClusterSeed(['R2',[2,3],2]); S23
     A seed for a cluster algebra of rank 2 of type ['R2',[2,3],2]
```

Here, `'R2'` refers to “rank 2”, `[2,3]` gives the parameters. For an explanation of the final 2, we refer to Section 6.2. Notice that if instead we let $b = 1$ and $c = 1$, we obtain

```
sage:  S11 = ClusterSeed(['R2',[1,1],2]); S11
        A seed for a cluster algebra of rank 2 of type ['A',2]
```

[1]Technically, this is the definition for a seed of a cluster algebra of geometric type. We give a more general definition of cluster algebra seeds in the next section.

We will see more examples of this phenomenon in a moment, but the point is that when $(b, c) = (1, 1)$, the associated cluster algebra is of "type A_2".

Let us keep working with the cluster seed $S11$ at the moment. We can see the B-matrix and initial cluster corresponding to this seed quite easily.

```
sage:  S11.cluster()
```

$$[x_0, x_1]$$

```
sage:  S11.b_matrix()
```

$$\begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}$$

Using this data, it is possible to construct the other generators of $\mathcal{A}(1, 1)$ by applying a sequence of exchanges. We define mutation in general down below. For now, let us mention that if we start with the initial cluster $[x_0, x_1]$, and mutate in the 0th direction, we replace the x_0 with x_2, defined as $x_2 = \frac{x_1+1}{x_0}$. This gives us a new seed, whose cluster is $[x_2, x_0]$.

```
sage:  S11.mutate(0); S11.cluster()
```

$$\left[\frac{x_1+1}{x_0}, x_1\right]$$

The exchange matrix of this new seed is simply $-B$.

```
sage:  S11.b_matrix()
```

$$\begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}$$

We can continue this procedure, and now mutate in the 1st direction, letting $x_3 = \frac{x_2+1}{x_1}$ replace x_1.

```
sage:  S11.mutate(1); S11.cluster()
```

$$\left[\frac{x_1+1}{x_0}, \frac{x_0+x_1+1}{x_0 x_1}\right]$$

```
sage:  S11.b_matrix()
```

$$\begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}$$

Notice that after this mutation, the exchange matrix of the obtained seed is B again. Consequently, we can iterate this procedure, applying the `mutate` command over and over. If we want to do this more efficiently, we can as well call `mutate` with a list of indices to apply from left to right.

```
sage:  S11.mutate([0,1,0,1])
```

If we are not only interested in the final seed, we can instead use the procedure `mutation_sequence`. Before doing that, we reset the cluster to the initial sequence of variables (in the initial seed).

```
sage:  S11.reset_cluster();
```

```
sage: S11.mutation_sequence([0,1,0,1,0],return_output='matrix')
```

$$\left[\begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}, \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}, \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}, \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}\right]$$

```
sage: S11.reset_cluster();
sage: S11.mutation_sequence([0,1,0,1,0],return_output='var')
```

$$\left[\frac{x_1+1}{x_0}, \frac{x_0+x_1+1}{x_0x_1}, \frac{x_0+1}{x_1}, x_0, x_1\right]$$

Here, the first command returns the sequence of exchange matrices obtained from this sequence of mutations, including the initial one. Notice, the sequence is exactly $[B, -B, B, -B, B, -B]$. The second command returns the list of cluster variables encountered as these exchanges occur. In the rank two case, this list is equivalent to $[x_2, x_3, x_4, x_5, x_6]$ corresponding to the $(b, c) = (1, 1)$-sequence $\{x_n\}$ referred to above.

Notice, that we have already found an interesting pattern, that is after five exchanges, we have arrived back essentially[2] at the same seed with which we started. This is therefore known as a cluster algebra of *finite type* and *finite mutation type*. Both of these concepts will be described in more detail below.

For our next example, we look at the $(b, c) = (2, 2)$ case, again a rank two cluster algebra.

```
sage: S22 = ClusterSeed(['R2',[2,2],2]); S22
      A seed for a cluster algebra of rank 2 of type ['A',[1,1],1]
```

Here again, notice that this specific rank two cluster algebra is recognized. In this case, this is our notation for a cluster algebra of affine type $\widetilde{A}_{1,1}$. We again, run the procedure `mutation_sequence`, and obtain the following:

```
sage: S22.mutation_sequence([0,1,0,1,0],return_output='var')
```

$$\left[\frac{x_1^2+1}{x_0}, \frac{x_1^4+x_0^2+2x_1^2+1}{x_0^2x_1}, \frac{x_1^6+x_0^4+2x_0^2x_1^2+3x_1^4+2x_0^2+3x_1^2+1}{x_0^3x_1^2},\right.$$

$$\frac{x_1^8+x_0^6+2x_0^4x_1^2+3x_0^2x_1^4+4x_1^6+3x_0^4+6x_0^2x_1^2+6x_1^4+3x_0^2+4x_1^2+1}{x_0^4x_1^3},$$

$$\left.\frac{x_1^{10}+x_0^8+2x_0^6x_1^2+3x_0^4x_1^4+4x_0^2x_1^6+5x_1^8+4x_0^6+9x_0^4x_1^2+12x_0^2x_1^4+10x_1^6+6x_0^4+12x_0^2x_1^2+10x_1^4+4x_0^2+5x_1^2+1}{x_0^5x_1^4}\right]$$

Unlike the previous case, the cluster variables appear to be getting more and more complicated, and that pattern continues. To understand these expressions better, we plug in the value 1 for x_0 and x_1.

```
sage: [cv.subs(S.x(0):1,S.x(1):1) for cv in ms]
                     [2, 5, 13, 34, 89]
```

From this data, one might conjecture, and it is in fact true, that the sequence

$$\{x_n : x_nx_{n-2} = x_{n-1}^2 + 1 \text{ and } x_0 = x_1 = 1\}$$

is precisely the Fibonacci numbers with even index.

[2]To be precise, this seed uses matrix $-B$ (equivalently B^T) instead of B, but these seeds are the same "up to equivalence", see Remark 3.4.

It is also clear that the cluster variables x_n's obtained by an instance of the (b, c)-sequence are rational functions in the indeterminates x_0 and x_1. More surprisingly, in spite of the divisions appearing, all such x_n's are actually *Laurent* polynomials, i.e. in the ring $\mathbb{Z}[x_0^{\pm 1}, x_1^{\pm 1}]$. This is actually a special case of one of the first major results in the theory of cluster algebras.

Theorem 1.1 (Laurent Phenomenon [FZ02a, FZ02b]). *Given any cluster algebra $\mathcal{A}$, which is parameterized by a choice of exchange pattern, a choice of coefficients (whose group ring is given as $\mathbb{ZP}$) and a choice of initial cluster $\{x_0, x_1, x_2, \ldots, x_{n-1}\}$ of generators, then all other generators, i.e. cluster variables, are Laurent polynomials in the ring $\mathbb{ZP}[x_0^{\pm 1}, x_1^{\pm}, \ldots, x_{n-1}^{\pm 1}]$.*

In the same paper in which Fomin and Zelevinsky prove this Laurent phenomenon, they made the following *positivity conjecture*.

Conjecture 1.2 (Positivity Conjecture). Given a cluster algebra $\mathcal{A}$ with an arbitrary exchange pattern, choice of coefficients $\mathbb{P}$, and an arbitrary initial cluster $\{x_0, x_1, \ldots, x_{n-1}\}$, then every generator of $\mathcal{A}$ can be written in

$$\mathbb{Z}_{\geq 0}\mathbb{P}[x_0^{\pm 1}, x_1^{\pm 1}, \ldots, x_{n-1}^{\pm 1}].$$

In other words, the Laurent expansions for cluster variables can be written using *positive* coefficients.

Positivity of the coefficients is significant, as it is conjecturally related to total-positivity properties of dual canonical bases [FZ99, FZ00, Zel02]. Nonetheless, this conjecture is still open despite nearly a decade of work by many researchers proving it for certain families of cluster algebras. Much of this work [CZ06, CK08, CR08, CP03, CS04, DiFK08, Dup09, Qin, MP07, MS08, MSW09, Nak09, Pro08, Sch08, ST09, SZ04, Spe07, Zel07] has been accomplished by exploration of examples, either by hand or by computer. As patterns to the Laurent polynomial expansions of cluster variables were noticed, the positivity conjecture and explicit formulas have been proven for more and more cases. This software provides further tools for such explorations.

2. What is a cluster algebra?

In this section, we give a more general and complete definition of cluster algebras, and in the next one, we describe the connection between cluster algebras and quivers. We say that a cluster algebra $\mathcal{A}$ is of *rank* n if $\mathcal{A}$ is subalgebra of an *ambient field* $\mathbb{F}$ isomorphic to a field of rational functions in n variables. Algebras are typically defined by *generators* and *relations*, but in the case of cluster algebras, instead of being handed all the generators at once, you are instead handed a distinguished set of n of them along with a constructive algorithm that can be used to obtain a complete set of generators. Note, that in general, a cluster algebra is infinitely-generated, however, any element of this distinguished generating set can be reached in finite time.

This distinguished generating set is called the set of *cluster variables*, the first n of which are known as the *initial* cluster variables. Any set of algebraically independent cluster variables of maximal size has the same cardinality, namely n, and these n-subsets are known as *clusters*. Pairs of clusters $\mathbf{x}$, $\mathbf{x}'$ whose intersection

is of size $(n-1)$ are related to one another by a binomial exchange relation of the form

$$\mathbf{x}' = (\mathbf{x} - \{x_k\}) \cup x'_k \qquad \text{where} \qquad x_k x'_k = p^+ M^+ + p^- M^-.$$

In the second equation, p^+ and p^- belong to a *coefficient semifield* $\mathbb{P}$ (a semifield is a field missing additive inverses and is subtraction-free), and M^+, M^- are monomials in the elements of $\mathbf{x} - \{x\}$ which share no common factor. We let $\oplus$ denote addition in the semifield $\mathbb{P}$, and multiplication is denoted in the usual way.

Definition 2.1 (Skew-symmetrizable matrices)**.** An n-by-n matrix B is called *skew-symmetrizable* if there exists a diagonal integer matrix D with strictly positive entries on the diagonal such that DB is skew-symmetric.

There is an algorithmic way to determine whether a matrix is skew-symmetrizable, and to find the diagonal matrix D, see Section 6.1.

Definition 2.2 (Labeled Seed for a cluster algebra)**.** A *labeled seed* for a cluster algebra $\mathcal{A} = \mathcal{A}(\mathbf{x}, \mathbf{y}, B)$ is a triple $(\mathbf{x}, \mathbf{y}, B)$ where

- $\mathbf{x} = \{x_0, x_1, \ldots, x_{n-1}\}$ is a cluster of n algebraically independent elements of ambient field $\mathcal{F}$,
- $\mathbf{y} = \{y_0, y_1, \ldots, y_{n-1}\}$ is an n-tuple of coefficients, elements of the semifield $\mathbb{P}$, and
- B is an n-by-n matrix that is skew-symmetrizable.

A labeled seed can be *mutated* into another labeled seed $(\mathbf{x}', \mathbf{y}', B')$ and all other clusters of $\mathcal{A}$, hence all other cluster variables, can be reached by applying a sequence of such mutations.

Definition 2.3 (Mutation of labeled seeds)**.** If $\mathcal{A}$ is a cluster algebra of rank n and $(\mathbf{x}, \mathbf{y}, B)$ is a labeled seed of $\mathcal{A}$, then for any $k \in \{0, 1, \ldots, n-1\}$, there exists another labeled seed $\mu_k(\mathbf{x}, \mathbf{y}, B) = (\mathbf{x}', \mathbf{y}', B') = (\mu_k(\mathbf{x}), \mu_k(\mathbf{y}), \mu_k(B))$ defined as follows:

The cluster $\mathbf{x}' = \{x_0, x_1, \ldots, \widehat{x_k}, \ldots, x_{n-1}\} \cup \{x'_k\}$ where

$$x'_k = \left(y_k \prod_{b_{ik}>0} x_i^{b_{ik}} + \prod_{b_{ik}<0} x_i^{-b_{ik}} \right) \Big/ (y_k \oplus 1) x_k;$$

the coefficient tuple $\mathbf{y}' = (y'_0, y'_1, \ldots, y'_{n-1})$ is given by

$$y'_j = \begin{cases} y_j \; y_k^{\max(b_{kj},0)} (y_k \oplus 1)^{-b_{kj}} \text{ if } j \neq k, \\ 1/y_k \text{ if } j = k \end{cases} ;$$

and the matrix $B' = \left[b'_{ij} \right]$ is given by

$$b'_{ij} = \begin{cases} -b_{ij} & \text{if } i = k \text{ or } j = k, \\ b_{ij} & \text{if } b_{ik} b_{kj} \leq 0, \\ b_{ij} + b_{ik} b_{kj} & \text{if } b_{ik}, b_{kj} > 0, \text{ or} \\ b_{ij} - b_{ik} b_{kj} & \text{if } b_{ik}, b_{kj} < 0 \end{cases} .$$

We say that $\mu_k(\mathbf{x}, \mathbf{y}, B)$ is the mutation in the kth direction.

The following important observation ensures that mutation of a labeled seed is again a labeled seed.

Proposition 2.4 (Proposition 4.5 of [FZ02a])**.** *If B is a skew-symmetrizable matrix, then so is $\mu_k(B)$ for $0 \leq k \leq n-1$.*

Another helpful fact about mutation is that it is an involution, i.e. for any $0 \leq k \leq n-1$, $\mu_k(\mu_k(\mathbf{x}, \mathbf{y}, B)) = (\mathbf{x}, \mathbf{y}, B)$.

Definition 2.5 (Cluster Algebras of Geometric Type)**.** A cluster algebra of *geometric type* is one where the coefficient semifield $\mathbb{P}$ is a special choice which leads to a great simplification of the above formulas. In this case,

$$\mathbb{P} = Trop(u_0, u_1, u_2, \ldots, u_{m-1})$$

where the addition $\oplus$ in $Trop(u_0, u_1, u_2, \ldots, u_{m-1})$ is defined as

$$\prod_j u_j^{a_j} \oplus \prod_j u_j^{b_j} = \prod_j u_j^{\min(a_j, b_j)}.$$

In particular, note that if $y_i = \prod_j u_j^{a_j}$ where all $a_j \geq 0$, then $y_i \oplus 1 = 1$. With $\mathbb{P}$ given as this tropical semifield, the group ring $\mathbb{ZP}$ is simply the ring of Laurent polynomials $\mathbb{Z}[u_0^{\pm 1}, u_1^{\pm 1}, \ldots, u_{m-1}^{\pm 1}]$.

Remark 2.6. Letting $\mathbb{P} = Trop(u_0, u_1, \ldots, u_{m-1})$, a labeled seed for a cluster algebra of geometric type is simply given as a pair $(\mathbf{x}, B)$, as opposed to a triple $(\mathbf{x}, \mathbf{y}, B)$, where $\mathbf{x} = \{x_0, x_1, \ldots, x_{n-1}, u_0, u_1, \ldots, u_{m-1}\} = \{x_0, x_1, \ldots, x_{m+n-1}\}$ and B is an $(n+m)$-by-n matrix whose top n-by-n portion is skew-symmetrizable. This notation agrees with that of Section 1.

By abuse of notation, we refer to the entire set of $(n+m)$ variables as a cluster, Only the first n cluster variables are *exchangeable*, the last m of them are known as *frozen variables* and appear in every single cluster. These frozen variables encode the same data as the coefficients did in the more general case described above. The exchange rules for mutation instead look like the following:

$$x_k' x_k = \prod_{b_{ik} > 0} x_i^{b_{ik}} + \prod_{b_{ik} < 0} x_i^{-b_{ik}}$$

and the mutation rule for the B-matrix is unchanged except that we must mutate entires in the last m rows appropriately as well. This mutation of the last m rows exactly agrees with the mutation of coefficients $\mathbf{y}$ in the general definition. In particular, if we let $y_j = \prod_{0 \leq i \leq m} u_i^{b_{i+n,j}}$ for $0 \leq j \leq n-1$, then we can recover the coefficient tuple $\mathbf{y}$ from the second halves of $\mathbf{x}$ and B.

Remark 2.7. Since cluster algebras of *geometric type* are sufficient for many applications and all of the computations currently possible in the cluster algebra package, we henceforth discuss the theory in terms of cluster algebras only of geometric type. We shall say that $\mathcal{A} = \mathcal{A}(\mathbf{x}, B)$ is a cluster algebra of rank n (with m coefficients) if it is a subalgebra of an *ambient field* $\mathbb{F}$ isomorphic to a field of rational functions in $(n+m)$ variables, m of which are *frozen*. This is because the cluster algebra $\mathcal{A}$ is a subalgebra of $\mathbb{ZP}[x_0^{\pm 1}, \ldots, x_{n-1}^{\pm 1}]$, and if $\mathbb{ZP} = \mathbb{Z}[u_0^{\pm 1}, \ldots, u_{m-1}^{\pm 1}]$, then $\mathcal{A}$ can be thought of as a subalgebra of $\mathbb{Z}[x_0^{\pm 1}, \ldots, x_{n-1}^{\pm 1}, u_0^{\pm 1}, \ldots, u_{m-1}^{\pm 1}]$, where the $u_i^{\pm 1}$'s are simply extra generators of $\mathcal{A}$ in addition to the set of exchangeable cluster variables.

Note: We abuse notation and often refer to the frozen variables as *coefficients*; and we always denote frozen variables as y_0 through y_{m-1} rather than the x_{n+i} or u_i notations used above.

We close this section with some examples and more information on some basic commands.

```
sage:  B3 = matrix([[0,1,0],[-1,0,-1],[0,1,0]]);
sage:  S3 = ClusterSeed(B3); S3
                 A seed for a cluster algebra of rank 3
```

Notice that unlike the earlier examples, the description of the seed does not include the type. This is because the input was only the matrix, and sage will not attempt to recognize the type unless it is asked for by the user or by a method.

```
sage:  S3.cluster()
```

$$[x_0, x_1, x_2]$$

```
sage:  S3.mutate(0)
sage:  S3.b_matrix()
```

$$B' = \begin{pmatrix} 0 & -1 & 0 \\ 1 & 0 & -1 \\ 0 & 1 & 0 \end{pmatrix}$$

```
sage:  S3.cluster()
```

$$\left[\frac{x_1+1}{x_0}, x_1, x_2\right]$$

We have therefore obtained a new labeled seed $(\mathbf{x}', B')$ by mutating in the 0th direction. Note that by default, `S3.mutate(0)` acted on and changed the object `S3` in place. There is an option to leave `S3` alone and just return the new object as a new output. If this behavior is desired, the command would be

```
sage:  S3new = S3.mutate(0,inplace=False)
```

Since mutation is an involution, if we mutate again in the 0th direction, we would recover the original labeled seed. So we instead mutate in a different direction.

```
sage:  S3.mutate(1)
sage:  S3.b_matrix()
```

$$B'' = \begin{pmatrix} 0 & 1 & -1 \\ -1 & 0 & 1 \\ 1 & -1 & 0 \end{pmatrix}$$

```
sage:  S3.cluster()
```

$$\left[\frac{x_1+1}{x_0}, \frac{x_0x_2+x_1+1}{x_0x_1}, x_2\right]$$

Let us explain why the second element (the element x_1') of this cluster is now $\frac{x_0x_2+x_1+1}{x_0x_1}$. This came from the exchange relation

$$x_1x_1' = x_2 + x_0',$$

which we read off of the second column of the exchange matrix $B' = \mu_0(B)$. Here $x_0' = \frac{x_1+1}{x_0}$ and so we obtain the desired Laurent polynomial in terms of the initial cluster variables x_0, x_1, and x_2 by plugging in for x_0' and simplifying.

For one more example of the exchange relation, let us now mutate in the 0th direction again. This corresponds to reading the first column of $B'' = \mu_1(\mu_0(B))$ which gives us the exchange relation $x_0'' = \frac{x_2+x_1'}{x_0'}$. Plugging in the relevant Laurent polynomials for x_0' and x_1', and dividing, we get a surprising cancellation and x_0'' is a Laurent polynomial:

$$x_0'' = \left(x_2 + \frac{x_0x_2+x_1+1}{x_0x_1}\right) \Big/ \left(\frac{x_1+1}{x_0}\right) = \frac{x_0x_1x_2+x_0x_2+x_1+1}{x_1(x_1+1)} = \frac{x_0x_2+1}{x_1}.$$

Using `sage`, we see

```
sage:  S3.mutate(0)
sage:  S3.b_matrix()
```

$$\begin{pmatrix} 0 & -1 & 1 \\ 1 & 0 & 0 \\ -1 & 0 & 0 \end{pmatrix}$$

```
sage:  S3.cluster()
```

$$\left[\frac{x_0x_2+1}{x_1}, \frac{x_0x_2+x_1+1}{x_0x_1}, x_2\right]$$

One last note about mutation, we can compress the above steps as the command

```
sage:  S3 = ClusterSeed(B3); S3.mutate([0,1,0]).
```

In other words, if a list is used at the input to `S3.mutate`, then the seed is mutated to a new seed by applying the sequence of mutations in the same order as given by the list.

At this point, `S3` is a labeled seed with matrix B'' and cluster $\mathbf{x}''$ as given. However, since a labeled seed is a choice of both an exchange matrix and a cluster, we also have methods to change the cluster. The first one is

```
sage:  S3.reset_cluster()
```

This command resets the cluster to the initial cluster $[x_0, x_1, \ldots, x_{n-1}]$ while leaving the exchange matrix alone. After running `S3.reset_cluster()`, we compute the exchange matrix and cluster, and obtain:

```
sage:  S3.b_matrix()
```

$$\begin{pmatrix} 0 & -1 & 1 \\ 1 & 0 & 0 \\ -1 & 0 & 0 \end{pmatrix}$$

```
sage:  S3.cluster()
```

$$[x_0, x_1, x_2]$$

A related command is `S3.set_cluster()` which lets the user set the initial cluster to be whatever they like. Note that in `sage`, arbitrary expressions in terms of indeterminates are not defined. However, integers (or even rational numbers) are fair to be plugged in. Additionally, if a rational function in terms of x_0 through x_{n-1} is desired, this can be accomplished by the commands `S3.x(0)` through `S3.x(n-1)`.

```
sage:  S3.set_cluster([7,11,13]); S3.b_matrix()
```

$$\begin{pmatrix} 0 & -1 & 1 \\ 1 & 0 & 0 \\ -1 & 0 & 0 \end{pmatrix}$$

```
sage: S3.cluster()
```

$$[7, 11, 13]$$

```
sage: S3.mutate([0,1,2,0]); S3.cluster()
```

$$[8/11, 115/77, 192/1001]$$

Note that at first glance, this might seem to falsify the Laurent Phenomenon, but it is actually allowed because all cluster variables are supposed to be Laurent polynomials in terms of the initial cluster variables. Since the integers 7, 11, and 13 are initial cluster variables, they are allowed to appear in the denominator.

```
sage: S3.b_matrix()
```

$$\begin{pmatrix} 0 & 1 & 0 \\ -1 & 0 & 1 \\ 0 & -1 & 0 \end{pmatrix}$$

```
sage: S3.set_cluster([S3.x(0)+S3.x(1),S3.x(1)^2,S3.x(0)/S3.x(2)])
sage: S3.cluster()
```

$$\left[x_0 + x_1, x_1^2, \frac{x_0}{x_2}\right]$$

```
sage: S3.mutate([0,1,0,2,0])
sage: S3.cluster()
```

$$\left[\frac{x_1^2 + 1}{x_0 + x_1}, \frac{x_0 x_1^2 + x_0 x_2 + x_1 x_2 + x_0}{x_0 x_1^2 x_2 + x_1^3 x_2}, \frac{x_0 x_1^2 x_2 + x_1^3 x_2 + x_0 x_1^2 + x_0 x_2 + x_1 x_2 + x_0}{x_0^2 x_1^2 + x_0 x_1^3}\right]$$

Again, these are Laurent polynomials in terms of the *initial* cluster variables obtained after setting them in this way.

Definition 2.8 (Principal coefficients)**.** An important cluster algebra of geometric type is one *with principal coefficients*. In this case, the initial exchange matrix B is $2n$-by-n and where the last n rows of this matrix is a rank n identity matrix.

Cluster algebras with principal coefficients are fundamental, because as explained in [FZ07] by Fomin and Zelevinsky, the formula for cluster variables in a cluster algebra with general coefficients (including those not of geometric type) can be described as a simple algebraic transformation of the formulas obtained for cluster variables with principal coefficients. See Theorem 3.7 of [FZ07] for more details. In future versions of this package, it is anticipated that working with *F-polynomials* and *g-vectors* will be easier with relevant methods included.

A cluster algebra with principal coefficients can be constructed rather simply by the command `S3.principal_extension()`. Before demonstration, let us reset the cluster:

```
sage: S3.b_matrix()
```

$$\begin{pmatrix} 0 & 1 & 0 \\ -1 & 0 & 1 \\ 0 & -1 & 0 \end{pmatrix}$$

```
sage: S3.reset_cluster(); S3.cluster()
```

$$[x_0, x_1, x_2]$$

Now, we demonstrate working with principal coefficients.

```
sage: SP3 = S3.principal_extension(); SP3
    A seed for a cluster algebra of rank 3 with 3 frozen variables
sage: SP3.b_matrix()
```

$$\begin{pmatrix} 0 & 1 & 0 \\ -1 & 0 & 1 \\ 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

```
sage: SP3.cluster()
```

$$[x_0, x_1, x_2, y_0, y_1, y_2]$$

Notice, that unlike the `mutate` command, which is a verb, `S3` is unaffected by the operation `SP3 = S3.principal_extension()`.

```
sage: S3.b_matrix()
```

$$\begin{pmatrix} 0 & 1 & 0 \\ -1 & 0 & 1 \\ 0 & -1 & 0 \end{pmatrix}$$

Let us try an example of mutating in this cluster algebra with principal coefficients. Here we use the command `SP3.mutation_sequence()` instead with optional arguments `return_output` which is either set to `'matrix'` or `'var'`.

```
sage: SP3.mutation_sequence([0,1,0,2],return_output='matrix')
```

$$\left[\begin{pmatrix} 0 & 1 & 0 \\ -1 & 0 & 1 \\ 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 0 & -1 & 0 \\ 1 & 0 & 1 \\ 0 & -1 & 0 \\ -1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 0 & 1 & 0 \\ -1 & 0 & -1 \\ 0 & 1 & 0 \\ 0 & -1 & 1 \\ 1 & -1 & 1 \\ 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 0 & -1 & 0 \\ 1 & 0 & -1 \\ 0 & 1 & 0 \\ 0 & -1 & 1 \\ -1 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 0 & -1 & 0 \\ 1 & 0 & 1 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \\ -1 & 1 & -1 \\ 0 & 1 & -1 \end{pmatrix}\right]$$

```
sage: SP3.mutation_sequence([0,1,0,2],return_output='var')
```

$$\left[\frac{x_1 + y_0}{x_0}, \frac{x_0y_0y_1 + x_1x_2 + x_2y_0}{x_0x_1}, \frac{x_0y_1 + x_2}{x_1}, \frac{x_0x_1y_0y_1y_2 + x_0y_0y_1 + x_1x_2 + x_2y_0}{x_0x_1x_2}\right]$$

A few words about this procedure.

(1) The command `SP3.mutation_sequence()` does not affect the object `SP3`, only returns the results of mutating in this order. If one wants actual seeds to work with rather than simply an output of matrices or cluster variables, one should use the option `return_output='seed'` (or omit this optional parameter since this is the default setting).

```
sage: seeds3 = SP3.mutation_sequence([0,1,0,2]); seeds3
```

```
[A seed for a cluster algebra of rank 3 with 3 frozen variables,
 A seed for a cluster algebra of rank 3 with 3 frozen variables,
  A seed for a cluster algebra of rank 3 with 3 frozen variables,
   A seed for a cluster algebra of rank 3 with 3 frozen variables,
    A seed for a cluster algebra of rank 3 with 3 frozen variables]
```

(2) With the optional parameters for returning output, the other options are `'matrix'` or `'var'`. The option `matrix` is self-explanatory. The option `var` outputs the new cluster variable at each step. The rest of the cluster variables in the associated clusters are suppressed, since otherwise a lot of redundant information would be printed or saved.

To return the rank (i.e. the number of exchangeable variables or columns in the exchange matrix B), one can simply use the command `SP3.n()`. The set of exchangeable variables can also be obtained by `SP3.exchangeable_variables()`. To return the number of coefficients or frozen variables (also equal to the number of rows minus the number of columns in B), we use the command `SP3.m()`. The frozen variables are obtained by `SP3.frozen_variables()`.

Not surprisingly, if we mutate `SP3` in place with the same sequence, it equals the last seed returned in the sequence.

```
sage: SP3.mutate([0,1,0,2]); SP3.cluster()
```

$$\left[\frac{x_0y_1 + x_2}{x_1}, \frac{x_0y_0y_1 + x_1x_2 + x_2y_0}{x_0x_1}, \frac{x_0x_1y_0y_1y_2 + x_0y_0y_1 + x_1x_2 + x_2y_0}{x_0x_1x_2}, y_0, y_1, y_2\right]$$

```
sage: SP3 == seeds3[len(seeds3)-1]
```

```
True
```

Notice that it is because of `sage`'s indexing starting at zero that the last seed is indexed by `len(seeds3)-1`, where `len` stands for "length". One can also access the last entry by `seeds3[-1]`.

It is also rather simple to strip off the frozen variables and obtain the coefficient-free cluster algebra by the command `SP3.principal_restriction()`. Like the command `S3.principal_extension()`, this does not change the object in place, and only returns a new object where only the top half of the matrix and the first n cluster variables are kept.

Warning: Since we did non-trivial mutations before restricting, we have a labeled seed where B is a 3-by-3 matrix but the associated cluster is Laurent polynomials in terms of y_0, y_1, and y_2, as well as x_0, x_1, and x_2.

```
sage:  SPR3 = SP3.principal_restriction(); SPR3.cluster()
```

$$\left[\frac{x_0y_1 + x_2}{x_1}, \frac{x_0y_0y_1 + x_1x_2 + x_2y_0}{x_0x_1}, \frac{x_0x_1y_0y_1y_2 + x_0y_0y_1 + x_1x_2 + x_2y_0}{x_0x_1x_2}\right]$$

This causes a problem when we try to mutate:

```
sage:  SPR3.mutate(0); SPR3.cluster()
```

$$\left[\frac{x_0y_0y_1 + x_0x_1 + x_1x_2 + x_2y_0}{x_0^2y_1 + x_0x_2}, \frac{x_0y_0y_1 + x_1x_2 + x_2y_0}{x_0x_1}, \frac{x_0x_1y_0y_1y_2 + x_0y_0y_1 + x_1x_2 + x_2y_0}{x_0x_1x_2}\right]$$

since the exchange relation should have involved y_i's but was truncated, and so we did not get the expected cancellation. Compare with

```
sage:  SP3.mutate(0); SP3.cluster()
```

$$\left[\frac{x_1 + y_0}{x_0}, \frac{x_0y_0y_1 + x_1x_2 + x_2y_0}{x_0x_1}, \frac{x_0x_1y_0y_1y_2 + x_0y_0y_1 + x_1x_2 + x_2y_0}{x_0x_1x_2}, y_0, y_1, y_2\right]$$

Thus it is **recommended** to run `SPR3.reset_cluster()` or `SPR3.set_cluster()` before actually restricting to the exchangeable cluster variables by running `SPR3 = SP3.principal_restriction()`.

3. Using quivers as cluster algebra seeds

In this section, we introduce a second way to input a cluster algebra seed. This uses the language of *quivers*, which is a fancy way of saying a directed (or oriented) graph. The term *quiver* originates in representation theory, where it was introduced by P. Gabriel at the beginning of the seventies. Gabriel wanted to emphasize the difference between the representation-theoretic and the graph-theoretic aspects of one and the same notion. For a quick introduction to quiver representations, please see references such as Section 5 of [Kel2]. An in-depth treatment is given, for example in the book by Assem, Simson, and Skowronski [ASS06]. The theory of quivers is an important one in representation theory, where fundamental questions come from studying the path algebra associated to such a directed graph.

For our purposes, we mostly use the quivers for bookkeeping purposes and thinking of them simply as directed graphs will be sufficient for most of our applications. In this package, a class of objects has been included as a placeholder for future development. For example, it is planned that in future versions of this package, some of the methods for quiver representations, as in preparation by Franco Saliola, will be available from this class as well. In the meantime, we will define what we need from quiver theory and describe the methods available in the current package as relevant to cluster algebra theory.

Definition 3.1. A *quiver* Q is a directed graph. We will only work with quivers on a finite number of vertices and which contains no loops (1-cycles) or 2-cycles. However, we do allow our quivers to have multiple edges between a pair of vertices, but since there are no 2-cycles, this means that all edges between two vertices must have the same direction. In general there is no restriction against oriented cycles on ≥ 3 vertices.

Definition 3.2 (Constructing an exchange matrix from a quiver)**.** Given a quiver Q on vertices v_0, v_1, through v_{n-1}, we let $\pm b_{ij}$ denote the number of edges between v_i and v_j. We let this number be positive if the edges are oriented from v_i to v_j and negative otherwise. We construct $B_Q = [b_{ij}]$ as the associated n-by-n matrix.

Definition 3.3 (Constructing a *pair-weighted* quiver from an exchange matrix)**.** To get a quiver Q_B from an $(m+n)$-by-n exchange matrix B is the reverse of the above construction, however, there are two nuances to emphasize.

1. For a cluster algebra seed to correspond to a quiver, the corresponding matrix B must satisfy $b_{ij} = -b_{ji}$ for all pairs $0 \leq i, j \leq n-1$. In other words, the top n-by-n portion of B must be skew-symmetric, not just skew-symmetrizable. Since cluster algebras for non-skew-symmetric seeds are also quite prevalent in the literature, our package works with a slight generalization of quivers, which we call *pair-weighted quivers*.

   We do not allow parallel edges in such quivers, and instead, we label each directed edge as an ordered pair $[b_{ij}, b_{ji}]$ such that the associated edge is oriented from v_i to v_j. Consequently, the first entry of each such pair is necessarily positive and the second is negative, but the direction of the edge must also be recorded. In the case that $b_{ji} = -b_{ij}$, i.e. the case of parallel edges, this label is simplified to be simply the positive number b_{ij}. We also omit the label $b_{ij} = 1$ when displaying graphics to make pictures easier to view.

   Note, that this notation differs from that in places such as [FZ03b] or [FST10], but is necessarily for precise computations. Our notation is inspired by Dlab-Ringel [DR76] and Dupont-Pérotin [DP10].
2. If $m > 0$, i.e. the matrix has more rows than columns, and for any b_{ij} where $i \geq n$, there is no b_{ji} in the matrix and so we do not have to worry about checking skew-symmetry for such entries. However, such vertices v_i correspond to a frozen variable and so we designate these vertices accordingly as "frozen vertices" to remind the user not to mutate or apply exchanges at such vertices.

Remark 3.4. This immediate connection between quivers and exchange matrices explains why we often consider exchange matrices up to simultaneous row and column permutations: two quivers are considered to be isomorphic if they are isomorphic as unlabeled digraphs, and this corresponds to considering exchange matrices up to simultaneous row and column permutations. The isomorphism reflects the fact that, as the cluster of an initial cluster seed $(\{x_1, \ldots, x_n\}, B)$ is invariant under permuting the variable indices, the cluster algebra does not depend on the ordering of the vertices in the corresponding quiver.

Definition 3.5 (Quiver Mutation)**.** While a quiver Q can be mutated in any of the n directions by constructing the associated exchange matrix B_Q, applying μ_k and then pulling back to the quiver $Q_{\mu_k(B)} = \mu_k(Q)$, there is also a three step

process that allows for a a nice visual description of quiver mutation (in the case of skew-symmetric B's).

(1) Reverse the direction of every oriented edge incident to vertex v_k. Call the resulting quiver Q'.
(2) For any 2-path $v_i \to v_k \to v_j$ that went through v_k in the original quiver Q, add a directed edge $v_i \to v_j$ in Q'. In other words, for any pair of vertices, $\{v_i, v_j\}$, if there are b_{ik} parallel edges from v_i to v_k and b_{kj} parallel edges from v_k to v_j, then in Q', we add $b_{ik}b_{kj}$ directed edges between v_i and v_j.
(3) In step 2, a 2-cycle may have been created, so the last step is to pair off and erase any such anti-parallel edges.

It is an easy exercise to see that the definition of matrix mutation $\mu_k(B)$ given in the previous section agrees with mutation of the quiver Q_B at vertex v_k. In the case of a pair-weighted quiver, it is easiest to mutate the associated matrix and then pull-back to a pair-weighted quiver.

Definition 3.6 (Mutation-equivalence)**.** Two quivers Q_1, Q_2 are said to be *mutation-equivalent* if one can be obtained from the other by a finite sequence of mutations, i.e., if there exists a finite sequence $i_1, \ldots, i_k$ such that $\mu_{i_k} \circ \cdots \circ \mu_{i_1}(Q_1) = Q_2$. The collection of all quivers mutation-equivalent to a given quiver Q is called *mutation class* of Q.

We now describe the numerous ways that a quiver can be constructed in our package. Firstly, a quiver can be constructed directly from an exchange matrix, or from a cluster seed in multiple ways.

```
sage:  B3 = matrix([[0,1,0],[-1,0,-1],[0,1,0]])
sage:  S3 = ClusterSeed(B3)
sage:  Q1 = Quiver(B3)
sage:  Q2 = Quiver(S3)
sage:  Q3 = S3.quiver()
sage:  Q1 == Q2; Q2 == Q3; Q1
            True        True        Quiver on 3 vertices
```

There are other possible constructors, such as from a directed graph:

```
sage:  dg = DiGraph()
sage:  dg.add_edges([[0,1],[2,1]])
sage:  Q4 = Quiver(dg)
sage:  Q1 == Q4
                            True
```

Warning: If one uses the digraph constructor, one must follow the conventions for that constructor as a sage object, in particular, digraphs do not allow multiple edges by default. For example, to get a quiver with parallel edges, one might be tempted to type

```
sage:  dg = DiGraph()
sage:  dg.allows_multiple_edges()
                            False
sage:  dg.add_edges([[0,1],[2,1],[2,1]])
sage:  dg.edges(labels=False)
                        [(0, 1), (2, 1)]
```

```
sage:  Q5 = Quiver(dg)
```

However, if one then asks

```
sage:  Q1 == Q5
                              True
```

as the multiple copies of edge $v_2 \to v_1$ are ignored. Instead, one should use the construction

```
sage:  dg = DiGraph()
sage:  dg.add_edges([[0,1,1],[2,1,2]])
sage:  Q6 = Quiver(dg)
sage:  Q1 == Q6; Q6.digraph().edges()
                 False        [(0,1,(2,-2)), (2,1,(1,-1))]
```

Note that all quivers are actually implemented as pair-weighted quivers, i.e. as a labeled digraph where multiple edges correspond to a pair $(b, -b)$ where $b \geq 2$. The program automatically converts the user's input with a single number indicating the edge label to a pair. A user can even label some edges as a single number, leave some edges unlabeled (as a single edge with pair-weight $(1, -1)$), and other edges as pairs; and the program will interpret this correctly. As mentioned above, multiple copies of an edge are ignored. More precisely, if they are given as labeled edges, then the label assigned is the one given to the last copy of the edge included.

```
sage:  dg = DiGraph()
sage:  dg.add_edges([[0,1,(1,-1)],[2,1,2]])
sage:  Q8 = Quiver(dg)
sage:  Q6 == Q8
                              True
```

A quiver can also be constructed more quickly by having sage do the intermediate work of constructing the digraph for you. Just simply type

```
sage:  Q9 = Quiver([[0,1,1],[2,1,2]])
```

or any of the analogous constructions described above for encoding the edges of a digraph (although again one should include edge labels instead of multiple copies of edges).

```
sage:  Q6 == Q9
                              True
```

You can also get a copy of a quiver already defined by a command such as

```
sage:  Q10 = Quiver(Q9)
sage:  Q10 == Q9
                              True
sage:  Q10.mutate(0)
sage:  Q10 == Q9
                              False
```

We did not emphasize it above, but a similar technique allows one to get a copy of a cluster seed. There is one other technique that can be used to construct a quiver,

or for that matter a cluster seed, but requires knowledge of quiver mutation types, and we leave the description of this construction to the next section.

We now introduce some of the possible methods our package contains for working with quivers along with associated examples. Most importantly, to get a picture of a quiver, we use the `show` command.

```
sage: Q1.show()
```

```
sage: Q1.mutate(1); Q1.show()
```

One quirk about the method `show` is that the graphic obtained can be a little random. If the placement of the vertices or the drawing of the graph is not optimal, it is recommended the user try running the `show` command again until the quiver renders in a more visually pleasing way.

Note that just as before, the command `Q1.mutate()` changes the object in place.

```
sage: Q1.b_matrix()
```

$$\begin{pmatrix} 0 & -1 & 0 \\ 1 & 0 & 1 \\ 0 & -1 & 0 \end{pmatrix}$$

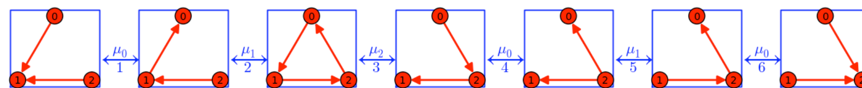
```
sage: Q1.mutate(1); Q1.b_matrix()
```

$$\begin{pmatrix} 0 & 1 & 0 \\ -1 & 0 & -1 \\ 0 & 1 & 0 \end{pmatrix}$$

A nice way to visualize a sequence of quiver mutations is the use of the command `mutation_sequence` with the optional parameter `show_sequence=True`. Unlike the `show` command, the quivers always render in the same circular way using this procedure so it is easier to compare vertices to one another.

```
sage: Q1.mutation_sequence([0,1,2,0,1,0],show_sequence=True)
```

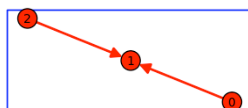
```
[Quiver on 3 vertices,
   Quiver on 3 vertices,
      Quiver on 3 vertices,
         Quiver on 3 vertices,
            Quiver on 3 vertices,
               Quiver on 3 vertices,
                  Quiver on 3 vertices]
```



Note that, here we are using the `mutation_sequence` on the quivers (rather than seeds), so the optional argument of `return_output` is not allowed. However, we can construct the associated cluster seed quite easily and then methods for viewing the associated quiver are still accessible, along with the other commands for cluster seeds.

```
sage:  NewS = ClusterSeed(Q1); Q2 = NewS.quiver()
sage:  Q2 == Q1; NewS.show()
```

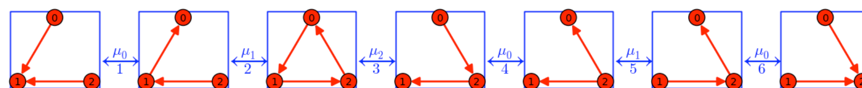
True



```
sage:  NewS.mutation_sequence([0,1,2,0,1,0],
         show_sequence=True,return_output='matrix')
```

$$\left[\begin{pmatrix} 0 & 1 & 0 \\ -1 & 0 & -1 \\ 0 & 1 & 0 \end{pmatrix}, \begin{pmatrix} 0 & -1 & 0 \\ 1 & 0 & -1 \\ 0 & 1 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 1 & -1 \\ -1 & 0 & 1 \\ 1 & -1 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & -1 \\ -1 & 1 & 0 \end{pmatrix},\right.$$

$$\left.\begin{pmatrix} 0 & 0 & -1 \\ 0 & 0 & -1 \\ 1 & 1 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 & -1 \\ 0 & 0 & 1 \\ 1 & -1 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ -1 & -1 & 0 \end{pmatrix}\right]$$



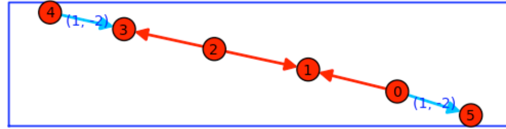
Another instructive command is `DiGraph` which lets the user construct the associated labeled directed graph encoding the quiver. Since `DiGraph` is already a class of objects, this allows the user access to a variety of other methods. One can then reconstruct a quiver with the altered directed graph, `dg` whenever desired, using the techniques described above, i.e. `Quiver(dg)`.

```
sage:  Quivs = Q1.mutation_sequence([0,1,2,0,1,0])
sage:  [Q.digraph().edges() for Q in Quivs]
     [[(0, 1, (1, -1)), (2, 1, (1, -1))],
        [(1, 0, (1, -1)), (2, 1, (1, -1))],
           [(0, 1, (1, -1)), (1, 2, (1, -1)), (2, 0, (1, -1))],
             [(0, 2, (1, -1)), (2, 1, (1, -1))],
                [(2, 0, (1, -1)), (2, 1, (1, -1))],
                  [(1, 2, (1, -1)), (2, 0, (1, -1))],
                     [(0, 2, (1, -1)), (1, 2, (1, -1))]]
```

Thus far, the examples included have been skew-symmetric and coefficient-free. We close this section with some examples which require pair-weighted quivers and frozen vertices.

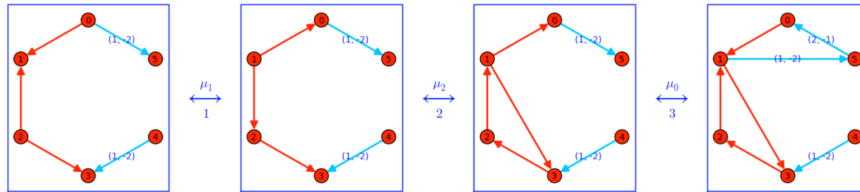
```
sage:  B = matrix([
         (0, 1, 0, 0, 0, 1),
```

```
          (-1, 0, -1, 0, 0, 0),
            (0, 1, 0, 1, 0, 0),
              (0, 0, -1, 0, -2, 0),
                (0, 0, 0, 1, 0, 0),
                  (-2, 0, 0, 0, 0, 0)])
sage:  S = ClusterSeed(B); S.show()
```



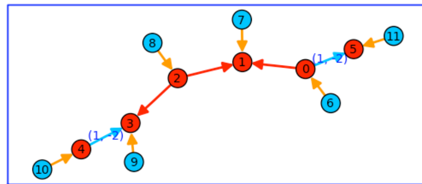

```
sage:  S.mutation_sequence([1,2,0], show_sequence=True,
         return_output='matrix')
```

$$\left[\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 1 \\ -1 & 0 & -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 & -2 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ -2 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & -1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 & -2 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ -2 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & -1 & 0 & 0 & 0 & 1 \\ 1 & 0 & -1 & 1 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 & 0 \\ 0 & -1 & 1 & 0 & -2 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ -2 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & -1 \\ -1 & 0 & -1 & 1 & 0 & 1 \\ 0 & 1 & 0 & -1 & 0 & 0 \\ 0 & -1 & 1 & 0 & -2 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 2 & -2 & 0 & 0 & 0 & 0 \end{pmatrix}\right]$$

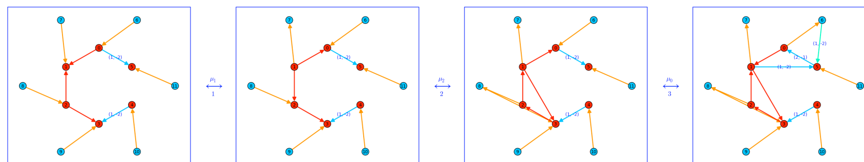


```
sage:  Q = Quiver(S)
sage:  Q2 = Q.principal_extension(); Q2; Q2.show()
```

```
          Quiver on 6 vertices with 6 frozen vertices
```



```
sage:  Q2.mutation_sequence([1,2,0],show_sequence=True)
```



If we instead produce a quiver by first producing the `principal_extension` of the cluster seed, and then constructing a quiver from it, we obtain an equal quiver as a result.

```
sage:  S2 = S.principal_extension()
sage:  Q3 = Quiver(S2); Q2 == Q3
                         True
```

Another way to work with quivers and cluster seeds is through the interactive mode available through the `sage-notebook`. This involves a command such as `S.interact()` or `Q.interact()`, as shown in Figure 1.

4. Finite type and finite mutation type classifications

So far we have described how a cluster algebra seed can be constructed from a skew-symmetrizable matrix or from a quiver. The last construction that we wish to discuss utilizes the notion of *quiver mutation types.* Before, we delve more into the specifics of this discussion, we begin with a few theoretical preliminaries.

Two natural questions that one can ask about a cluster algebra (or its seed) once the initial definitions have been given are the following:

Given a cluster algebra $\mathcal{A} = \mathcal{A}(\mathbf{x}_0, B_0)$, with initial seed $(\mathbf{x}_0, B_0)$,

- are there a finite number of generators (cluster variables) x for $\mathcal{A}$ as we take the union of all clusters $\mathbf{x}$ as we mutate?
- are there a finite number of exchange matrices B for $\mathcal{A}$ as we mutate into different seeds?

Definition 4.1. If there are a finite number of cluster variables for $\mathcal{A}$, we say that $\mathcal{A}$ is of *finite type.*

Definition 4.2. If there are a finite number of exchange matrices for $\mathcal{A}$, we say that $\mathcal{A}$ is of *finite mutation type.*

An important theorem that greatly simplifies our notation for geometric type is the following theorem by Gekhtman, Shapiro, and Vainshtein:

Theorem 4.3 (Theorem 7.4 of [GSV10])**.** *If $\mathcal{A}$ is a cluster algebra (i) of geometric type, or (ii) has nondegenerate exchange matrix, and $(\mathbf{x}, B)$, $(\mathbf{x}', B')$ are two seeds for $\mathcal{A}$, such that cluster $\mathbf{x}'$ is simply the permutation σ of cluster $\mathbf{x}$, then the exchange matrices B' and B must also be the same, up to simultaneous permutation of its rows and columns by the same σ. In particular, the* cluster determines the seed *in the above cases.*

From this theorem, it is clear that any cluster algebra of finite type must have a finite number of clusters, hence a finite number of seeds and exchange matrices.

Corollary 4.4 (Finite type implies finite mutation type)**.** *A cluster algebra of finite type* is also *of finite mutation type.*

However, the converse is false, the simplest counter-example being the rank two example $\mathcal{A}(2, 2)$ discussed in the Introduction.

Classifying cluster algebras of finite type was one of the first natural questions about cluster algebras, and led Fomin and Zelevinsky to the following beautiful theorem.

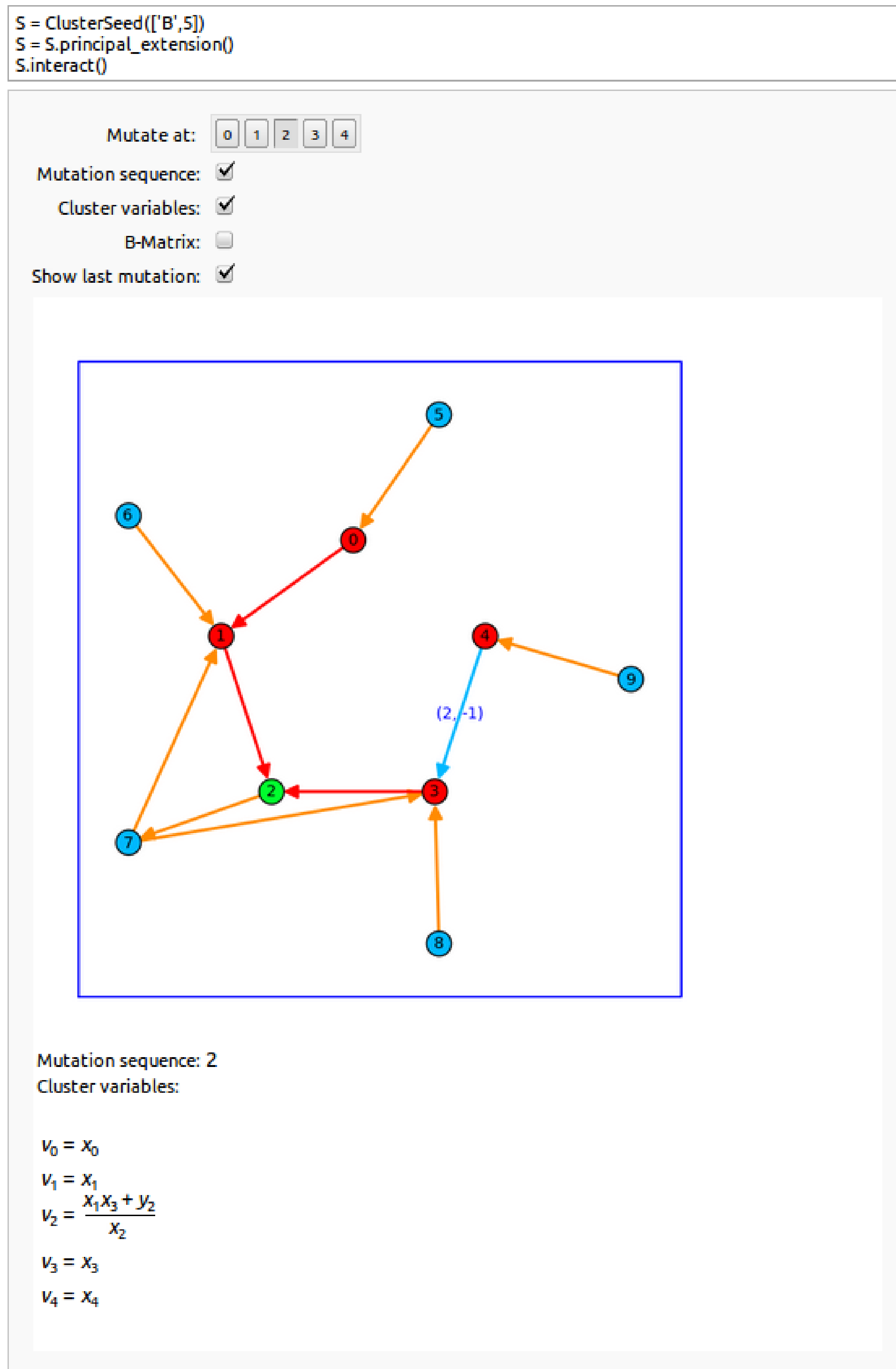


FIGURE 1. The interactive mode of the cluster package in the `sage-notebook`.

Theorem 4.5 (Theorem 1.5 of [FZ03b]). *The following three conditions about a cluster algebra $\mathcal{A} = \mathcal{A}(\mathbf{x}_0, B_0)$ are equivalent:*

- *Cluster algebra $\mathcal{A}$ is of finite type.*
- *In every seed $(\mathbf{x}, B)$ that is mutation-equivalent to $(\mathbf{x}_0, B_0)$, the exchange matrix B satisfies $|b_{ij}b_{ji}| \leq 3$ for all pairs $1 \leq i, j \leq n$.*
- *There exists a mutation-equivalent seed $(\mathbf{x}_1, B_1)$ such that the exchange matrix B_1 is a* skew-symmetric version *of a Cartan matrix of a finite-dimensional Lie algebra*[3].

In particular, cluster algebras of finite type are given by the same Cartan-Killing classification as that describing Lie algebras via Dynkin diagrams:

$$A_n,\ B_n,\ C_n,\ D_n,\ E_6,\ E_7,\ E_8,\ F_4,\ \text{and } G_2.$$

Given a cluster algebra seed S for $\mathcal{A}$, it therefore makes sense to ask whether or not S is mutation-equivalent to a seed $(\mathbf{x}, B)$ where the exchange matrix B is a skew-symmetric version of the Cartan matrix of type A_n (resp. B_n, C_n, D_n, E_6, E_7, E_8, F_4, or G_2). If so, we call $\mathcal{A}$ a cluster algebra of mutation type A_n (resp. B_n, C_n, D_n, E_6, E_7, E_8, F_4, or G_2). We also call all exchange matrices and the corresponding quivers of such a cluster algebra of mutation type A_n (resp. B_n, C_n, D_n, E_6, E_7, E_8, F_4, or G_2).

Our program has algorithms for identifying mutation types of exchange matrices and quivers. In the cases of the exceptional types, E_6, E_7, E_8, F_4 and G_2, it is sufficient to hard-code a catalog of the mutation classes. This is done to avoid recomputing the mutation class whenever checking a mutation type. In classical types however, the parameter n can be any positive integer, and we instead utilize theoretical results of [CCS06] (type A_n), [Stu11] (types B_n and C_n), and [Vat08] (type D_n) to identify them for any rank n.

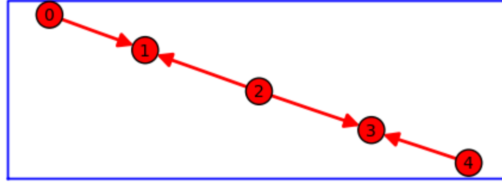
Recall that a quiver (resp. pair-weighted quiver) encodes the same information as a skew-symmetric (resp. skew-symmetrizable) matrix. To avoid duplication of data types, we have introduced a new class of objects known as quiver mutation types. Note that these can be implemented with or without brackets.

```
sage:  QM1 = QuiverMutationType(['A',5])
sage:  QM2 = QuiverMutationType('A',5); QM1 == QM2
                                   True
sage:  QM1
                                  ['A', 5]
sage:  type(QM1)
        <class 'sage.all_cmdline.QuiverMutationType_Irreducible'>
sage:  QM1.b_matrix()
```

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ -1 & 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 & -1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

[3] Given a Cartan matrix A, we make a skew-symmetric B_A by replacing the 2's on the diagonal with 0's, and picking a bipartite coloring of the Dynkin diagram associated to A so that $b_{ij} = |a_{ij}|$ if directed edge $v_i \to v_j$ would go from white to black, and $b_{ij} = -|a_{ij}|$ otherwise, see Section 5.

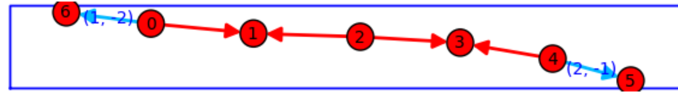
```
sage: Quiv = QM1.standard_quiver(); Quiv
          Quiver on 5 vertices of type ['A', 5]
sage: Quiv.show()
```




```
sage: QM2 = QuiverMutationType('BC',6,1); QM2
                    ['BC',5,1]
sage: QM2.b_matrix()
```

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ -1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 0 \\ -2 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

```
sage: QM2.standard_quiver().show()
```



Each quiver mutation type has a number of attributes and methods associated to it. We already saw an example of two key methods: `b_matrix` and `standard_quiver`, i.e. each quiver mutation type object encodes a specific canonical exchange matrix and the associated pair-weighted quiver. This characterizes only one representative out of the relevant possible mutation-class, but it is enough data to determine the appropriate cluster algebra seed up to mutation-equivalence. We hard-coded these representatives so that the associated quiver is an oriented Dynkin diagram such that each vertex is a sink or source. For future reference, such a quiver and seed is known as *bipartite*.

More generally, each of these representative quivers are *trees* and *acyclic*. Because of results from representation theory and otherwise, there are a number of results in cluster algebra theory that hold when the associated quiver is bipartite (resp. a tree or acyclic), but the result is incorrect, or a proof is unknown when the quiver lacks the relevant property. Here are some examples:

Theorem 4.6. [Nak09] *If a cluster algebra $\mathcal{A}$ is given by a seed that is mutation-equivalent to one which is skew-symmetric and bipartite, then all cluster variables of $\mathcal{A}$ have positive expansions as Laurent polynomials* [4].

---

[4] By theorems of Fan Qin [Qin] and an updated version of [Nak09], positivity has been proven for all skew-symmetric acyclic seeds.

Theorem 4.7 (Proposition 9.2 in [FZ03b]). *If Q is a quiver that is a tree as an undirected graph then Q is mutation-equivalent to any Q' where Q' has the same underlying undirected graph as Q but the edges of Q' are oriented arbitrarily*[5].

Theorem 4.8 (Corollary 1.21 in [BFZ05]). *Let $\mathcal{A} = \mathcal{A}(\mathbf{x}, B)$ be a cluster algebra where B corresponds to an acyclic seed. Let x'_i denote the unique element in cluster $\mu_i(\mathbf{x})$ which is not contained in $\mathbf{x}$. Then we have the following:*

- *$\mathcal{A}$ is finitely generated by the set $\chi = \{x_1, x'_1, \ldots, x_n, x'_n\}$,*
- *The standard monomials (those not containing the factor $x_i x'_i$ for any $i \in \{0, 1, \ldots, n-1\}$) in χ form a $\mathbb{ZP}$-basis of $\mathcal{A}$, and*
- *The binomial exchange relations involving $x_i x'_i$ on the left-hand-sides generate the ideal of relations among the generators χ.*

Because of the importance of these properties, and other related ones, there are methods to check whether a given cluster seed, quiver, or quiver mutation type satisfies them:

```
is_finite(), is_mutation_finite(), is_bipartite(), is_acyclic(),...
```

There are a few other checks that we have not explained yet, but we will provide an annotated list of all of the checkable properties in Section 6.

```
sage:  QM1.properties()
         ['A', 5] has rank 5 and the following properties:
                   - irreducible:  True
                 - mutation finite:  True
                  - simply-laced:  True
                 - skew-symmetric:  True
                     - finite:  True
                    - affine:  False
                   - elliptic:  False
sage:  QM2.properties()
       ['BC', 6, 1] has rank 7 and the following properties:
                   - irreducible:  True
                 - mutation finite:  True
                  - simply-laced:  False
                 - skew-symmetric:  False
                    - finite:  False
                     - affine:  True
                   - elliptic:  False
```

Most importantly, our program allows the user to construct a cluster seed or quiver by using a quiver mutation type. The associated quiver is the standard quiver that is hard-coded as a representative for each type; and the associated cluster seed is obtained from this choice of quiver.

```
sage:  ClusterSeed(['A',5])
        A seed for a cluster algebra of rank 5 of type ['A', 5]
sage:  ClusterSeed(['BC',6,1])
```

[5]Note: this list of mutation-equivalent quivers is not exhaustive, for example a quiver of type A_3 is both mutation-equivalent to any orientation of a path on three vertices; or to an oriented triangle.

```
    A seed for a cluster algebra of rank 7 of type ['BC', 6, 1]
sage: Quiver(['A',5])
              Quiver on 5 vertices of type ['A', 5]
sage: Quiver(['BC',6,1])
            Quiver on 7 vertices of type ['BC', 6, 1]
```

4.1. **Finite mutation type classification.** We now describe theoretical results regarding the classification of cluster algebras of finite mutation type. Again, we use the notation of pair-weighted quivers so our descriptions of some of the results will differ slightly from the work of Felikson-Shapiro-Tumarkin [FST10]. Our story begins however with Felikson-Shaprio-Tumarkin's first paper [FST08] which classified *skew-symmetric* cluster algebras of finite mutation type.

Theorem 4.9 (Theorem 6.1 of [FST08])**.** *The following two conditions about a cluster algebra $\mathcal{A} = \mathcal{A}(\mathbf{x}_0, B_0)$ with skew-symmetric B_0 are equivalent:*

- *$\mathcal{A}$ is of finite mutation type,*
- *$\mathcal{A}$ has one of the following properties:*
  1. *$\mathcal{A}$ is of rank* 2*,*
  2. *$\mathcal{A}$ is associated to a* cluster algebra corresponding to a surface*, or*
  3. *$\mathcal{A}$ is one of* 11 *exceptional types E_6, E_7, E_8, affine $\tilde{E}_6$, $\tilde{E}_7$, $\tilde{E}_8$, elliptic $\tilde{E}_6^{(1)}$, $\tilde{E}_7^{(1)}$, $\tilde{E}_8^{(1)}$, or one of two other types X_6 and X_7, which were found by Derksen and Owen* [DO08]*.*

Rank two cluster algebras were already described in the introduction, and are clearly mutation-finite since mutation of such an exchange matrix B simply leads to $-B$.

Describing cluster algebras of surfaces is beyond the scope of this compendium, however it is planned that future installments of this software will handle such cluster algebras and their description will be spelled out at that time. Please see Fomin, Shaprio, and D. Thurston's papers [FST08, FT08] for a description or [MSW09] where Schiffler, Williams, and the first author prove positivity of Laurent expansions for such cluster algebras. Nonetheless, we mention here that cluster algebras corresponding to *polygons with* 0*,* 1*, or* 2 *punctures, or to an annulus*, can also be described as the skew-symmetric types A_n, D_n, $\tilde{D}_n$, or $\tilde{A}_{r,s}$, respectively. The first two cases are of finite type and the second two are of affine type. Any other finite or affine type is of exceptional type or is not skew-symmetric. We illustrate corresponding representative quivers in the next section.

We have met some of the eleven exceptional types before, the types E_6, E_7, and E_8 are of finite type and thus of finite mutation type. We give representative quivers for the remaining eight in the next section. The affine types $\tilde{E}_6$, $\tilde{E}_7$, and $\tilde{E}_8$ each have a bipartitely oriented tree as a quiver representative; however the other five have no acyclic representatives.

4.2. **Skew-symmetrizable cluster algebra seeds of finite mutation type.** In cutting edge work this summer [FST10], Felikson-Shapiro-Tumarkin generalized their previous work to a classification including mutation-finite weighted quivers that are not skew-symmetric.

Theorem 4.10 (Theorems 2.8 and 5.13 of [FST10])**.** *The following three conditions about a cluster algebra $\mathcal{A} = \mathcal{A}(\mathbf{x}_0, B_0)$ with skew-symmetrizable B_0 are equivalent:*

- *$\mathcal{A}$ is of finite mutation type,*
- *In every seed $(\mathbf{x}, B)$ that is mutation-equivalent to $(\mathbf{x}_0, B_0)$, the exchange matrix B satisfies $|b_{ij}b_{ji}| \leq 4$ for all pairs $1 \leq i, j \leq n$.*
- *$\mathcal{A}$ has one of the following properties:*
  1. *$\mathcal{A}$ is of rank 2,*
  2. *$\mathcal{A}$ is decomposable into* blocks*, as described in* [FST10]*, or*
  3. *$\mathcal{A}$ is one of the 11 exceptional types in Theorem 4.9 or one of the 7 exceptional types $\tilde{G}_2, F_4, \tilde{F}_4, V_4, W_4, Y_4$, and Z_6.*

Remark 4.11. One can get from our notation of pair-weighted quivers to the notion of weighted quivers in [FST10] by the following: if an edge of our quiver has the pair-weight $[b, -c]$, then the corresponding weight in their notation is bc. While their notation has several advantages and simplifies the statements of certain theorems, for computations it obscures the differences between different mutation classes. For example, cluster algebras of types B_n and C_n would have the same weighted quivers. Even though these cluster algebras give rise to the same cluster complexes (i.e. the clique complex induced by the graph whose vertices are seeds and whose edges are mutations), the Laurent expansions of cluster variables are quite different in these two cases.

To illustrate this example we introduce two new commands. See Section 4.4 for details on the associated algorithms:

1) Given a cluster algebra of finite mutation type, we can use the command `b_matrix_class` to obtain a list of all the exchange matrices that are mutation-equivalent to a given initial seed. To avoid extraneous duplication, we only output one matrix up to simultaneous permutation of rows and columns.

For example, in the B_3 versus C_3 cases, notice that the list of exchange matrices in the respective mutation classes are negative transposes of one another[6].

```
sage:  S3 = ClusterSeed(['B',3]); S3.b_matrix_class()
```

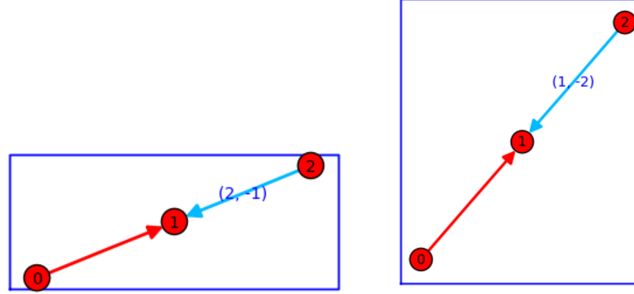
$$\left[\begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 2 \\ -1 & -1 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & -2 \\ -1 & 1 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 1 & 1 \\ -2 & 0 & 0 \\ -1 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 2 & 0 \\ -1 & 0 & 1 \\ 0 & -1 & 0 \end{pmatrix}, \begin{pmatrix} 0 & -1 & 1 \\ 2 & 0 & -2 \\ -1 & 1 & 0 \end{pmatrix}\right]$$

```
sage:  S4 = ClusterSeed(['C',3]); S4.b_matrix_class()
```

$$\left[\begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ -1 & -2 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & -1 \\ -1 & 2 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 2 & 1 \\ -1 & 0 & 0 \\ -1 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 1 & 0 \\ -2 & 0 & 1 \\ 0 & -1 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 1 & -1 \\ -2 & 0 & 1 \\ 2 & -1 & 0 \end{pmatrix}\right]$$

```
sage:  S3.show(); S4.show()
```

[6] This would be clearer if we included all mutation-equivalent matrices rather than just those up to permutation, which could be accomplished by `S3.b_matrix_class(up_to_equivalence=False)`. In particular the last matrices in both of these lists are negative transposes of each other if we also swap the first and second rows/columns.




```
sage:  S3.quiver().digraph().edges()
            [(0, 1, (1, -1)), (2, 1, (2, -1))]
sage:  S4.quiver().digraph().edges()
            [(0, 1, (1, -1)), (2, 1, (1, -2))]
```

There is an analogous command that works for cluster algebras of *finite type*:

2) The command `variable_class` will output the list of all cluster variables obtained as one mutates through all mutation-equivalent seeds.

```
sage:  S3.variable_class()
```

$$\left[x_0, x_1, x_2, \frac{x_1+1}{x_0}, \frac{x_0x_2^2+1}{x_1}, \frac{x_1+1}{x_2}, \frac{x_0x_2^2+x_1+1}{x_0x_1}, \frac{x_0x_2^2+x_1+1}{x_1x_2}, \frac{x_0x_2^2+x_1^2+2x_1+1}{x_0x_1x_2}, \frac{x_0x_2^2+x_1^2+2x_1+1}{x_1x_2^2}, \frac{x_1^3+x_0x_2^2+3x_1^2+3x_1+1}{x_0x_1x_2^2}, \frac{x_0^2x_2^4+3x_0x_1x_2^2+x_1^3+2x_0x_2^2+3x_1^2+3x_1+1}{x_0x_1^2x_2^2}\right]$$

```
sage:  S4.variable_class()
```

$$\left[x_0, x_1, x_2, \frac{x_1+1}{x_0}, \frac{x_0x_2+1}{x_1}, \frac{x_1^2+1}{x_2}, \frac{x_0x_2+x_1+1}{x_0x_1}, \frac{x_1^2+x_0x_2+1}{x_1x_2}, \frac{x_1^3+x_1^2+x_0x_2+x_1+1}{x_0x_1x_2}, \frac{x_0^2x_2^2+x_1^2+2x_0x_2+1}{x_1^2x_2}, \frac{x_0^2x_2^2+x_1^3+x_0x_1x_2+x_1^2+2x_0x_2+x_1+1}{x_0x_1^2x_2}, \frac{x_1^4+x_0^2x_2^2+2x_1^3+2x_0x_1x_2+2x_1^2+2x_0x_2+2x_1+1}{x_0^2x_1^2x_2}\right]$$

In conclusion, even though the quivers of type B_3 and C_3 look quite similar and they have the same cluster complex, the Laurent polynomials are quite different. For example, the bipartite seed for a cluster algebra of type B_3 leads to cluster variables whose numerators have degree 6, while the numerators are only of degree at most 4 in the case of C_3. Similar phenomena happen for other dual cluster algebras, e.g. types B_n versus C_n for $n \geq 3$, or pairs of seeds: $(\mathbf{x}, B)$ and $(\mathbf{x}, B^T)$. Here and below, we adapt the term "dual" from the notion for Kac-Moody algebras.

Nuances like these make the non-skew-symmetric cases more difficult to analyze. Nonetheless, using the classification (via folding of skew-symmetric quivers) appearing in [FST10], it has been possible to include descriptions of mutation classes for those classes that correspond to a non-simply laced Dynkin diagram of finite or

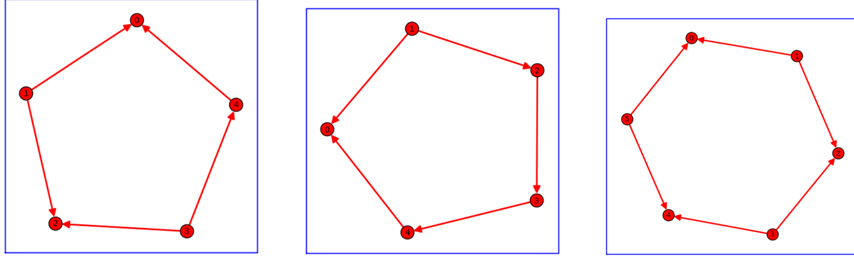
affine type, as well as the weighted quivers listed as exceptional cases in [FST10]. For the classification of non-simply laced affine Dynkin diagrams, we use the tables of Kac [Kac94, pgs. 53-55]. However, the notation here is not explicit enough either as a number of cluster algebra mutation classes are again collapsed together. We therefore follow notation of Dupont-Pérotin [DP10] instead. The Dupont-Pérotin notation specifies a quiver by indicating what the two ends look like, where the choices are that of a Dynkin diagram of type B, C or D. We say more about this notation in the next section. Since many users might be more familiar with the Kac-Moody notation, through careful coercing, if a user inputs a typical Kac-Moody type, it is recognized and translated into the appropriate notation that our software uses.

```
sage:  QuiverMutationType('C',2)
                                  ['B',2]
sage:  QuiverMutationType('B',4,1)
                                 ['BD',4,1]
sage:  QuiverMutationType('C',4,1)
                                 ['BC',4,1]
sage:  QuiverMutationType('A',2,2)
                                 ['BC',1,1]
sage:  QuiverMutationType('A',4,2)
                                ['BC', 2, 1]
sage:  QuiverMutationType('A',5,2)
                                ['CD', 3, 1]
sage:  QuiverMutationType('A',6,2)
                                ['BC', 3, 1]
sage:  QuiverMutationType('A',7,2)
                                ['CD', 4, 1]
sage:  QuiverMutationType('D',5,1)
                                  ['D',5,1]
sage:  QuiverMutationType('D',5,2)
                                 ['CC',5,1]
sage:  QuiverMutationType('D',4,3)
                                 ['G',2,-1]
sage:  QuiverMutationType('E',6,1)
                                  ['E',6,1]
sage:  QuiverMutationType('E',6,2)
                                 ['F',4,-1]
sage:  QuiverMutationType('F',4,1)
                                  ['F',4,1]
```

As for the finite types, our program has algorithms for identifying exchange matrices of affine types. In affine type $\tilde{A}_n$, we have a similar coercion issue in the case of simply-laced affine $\tilde{A}_{r,s}$ types where two parameters (rather than one parameter) is required to specify a mutation-equivalence type. This example is special because it is the only finite or affine type with a Dynkin diagram which is not a tree. Instead its Dynkin diagram is a cycle on n vertices, and here quivers Q_1

and Q_2 are only mutation-equivalent if they have the same number of edges oriented clockwise and the same number of edges oriented counter-clockwise. Actually, if all arrows are reversed, it is also the same type. The mutation classes of types $\tilde{A}_{r,s}$ can be classified using theoretical results in [Bas10].

```
sage:  Qu = Quiver(['A',[2,3],1]); Qu
           Quiver on 5 vertices of type ['A', [2, 3], 1]
sage:  Qu.show()
sage:  Quiver(['A',[4,1],1]).show()
sage:  Quiver(['A',[3,3],1]).show()
```



Notice also that the representative quiver for an affine $\tilde{A}_{r,s}$-type is made as bipartite as possible and that mutation type `['A',[r,s],1]` is coerced into type `['A',[s,r],1]` when $s < r$.

The remaining affine types can be found in Section 6.2 and are classified using results in [Hen09] and [Stu11].

Beside the described coercions, we also include some basic coercions such as letting type D_2 coerce into type $A_1 \times A_1$, D_3 coerce into A_3, C_2 coerce into B_2, small rank two examples $\mathcal{A}(b, c)$ coerce into A_2, B_2, G_2, and $\tilde{A}_{1,1}$, and $\tilde{BC}_1$ for $(b, c) = (1, 1), (1, 2), (1, 3), (2, 2)$, and $(1, 4)$, respectively. Here, $\tilde{BC}_1$ simply means the type `['BC',1,1]` which is a denegenerate version of the `['BC',n,1]` family of Dynkin diagrams used above. More technical details can be found in Section 6.2, including other families of types and more coercions.

4.3. Class sizes of finite and affine quiver mutation types.

In this section, we discuss the sizes of mutation classes of finite and affine types. Those results and conjectures are used to compute the size of mutation classes without explicitly computing the class. The *class size* of a cluster seed or quiver is defined to be the number of exchange matrices or quivers which are mutation-equivalent to the given cluster seed or quiver, respectively. Here, we consider seeds and quivers up to isomorphism.

Theorem 4.12 (Class sizes of finite types). *The number of exchange matrices or quivers of finite*

- *type* A_n [Tor08] *is given by*

$$\frac{1}{n+3}\left[\frac{1}{n+1}\binom{2n}{n} + \binom{n+1}{(n+1)/2} + \binom{2n/3}{n/3}\right],$$

where the second term is omitted if $(n+1)/2$ *is not integral and the third term if* $n/3$ *is not integral.*

- *type B_n or of type C_n* [Stu11] *is given by*
$$\frac{1}{n+1}\binom{2n}{n}.$$
- *type D_n* [BT09] *is for $n = 4$ given by 6, and for $n \geq 5$, it is given by*
$$\sum_{d|n} \frac{\phi(n/d)}{2n}\binom{2d}{d}.$$
- *types $E_6, E_7, E_8, F_4,$ and G_2 are given by $67, 416, 1574, 15,$ and 2.*

Theorem 4.13 ([BPRS10])**.** *The number of exchange matrices or quivers of affine type $\tilde{A}_{r,s}$ is given by*
$$\begin{cases} \frac{1}{2}\sum_{k|r,k|s} \frac{\phi(k)}{r+s}\binom{2r/k}{r/k}\binom{2s/k}{s/k} & \text{if } r \neq s, \\ \frac{1}{2}\left(\frac{1}{2}\binom{2r}{r} + \sum_{k|r} \frac{\phi(k)}{4r}\binom{2r/k}{r/k}^2\right) & \text{if } r = s. \end{cases}$$
where $\phi(k)$ is Euler's totient function, i.e., the number of $1 \leq d \leq k$ coprime to k.

Conjecture 4.14 ([Stu11])**.** The number of exchange matrices or quivers of affine

- type $\tilde{BB}_n$ or of type $\tilde{CC}_n$ is given by
$$\binom{2n-1}{n-1} + \binom{n-1}{n/2-1}$$
where the second term is omitted if n is odd.
- type $\tilde{D}_n$ is for $n = 4$ given by 9, and for $n \geq 5$, it is given by
$$2\binom{2n}{n} + \binom{n}{n/2},$$
where the second term is omitted if n is odd.
- type $\tilde{BC}_n$ is given by
$$\binom{2n}{n}.$$
- type $\tilde{BD}_n$ or of type $\tilde{CD}_n$ is given by
$$2\binom{2(n-1)}{n-1}.$$

Theorem 4.15. *The number of exchange matrices or quivers of*

- *affine types $\tilde{E}_6, \tilde{E}_7, \tilde{E}_8, \tilde{F}_4,$ and $\tilde{G}_2$ are given by $132, 1080, 7560, 60,$ and 6.*
- *elliptic types $\tilde{E}_6^{(1)}, \tilde{E}_7^{(1)},$ and $\tilde{E}_8^{(1)}$ are given by $49, 506,$ and 5739.*
- *the other exceptional mutation-finite types $V_4, W_4, X_6, X_7, Y_6,$ and Z_6 are given by $7, 2, 5, 2, 90,$ and 35.*

4.4. **Algorithms for computing mutation classes.** The four commands

`mutation_class`, `b_matrix_class`, `cluster_class`, `variable_class`

each utilize the auxiliary command obtained by adding `_iter`, which constructs an iterator that will run through all the objects in the corresponding mutation class. For quivers, there is only the method `mutation_class`. The first three methods are directly derived from `mutation_class_iter`, we therefore begin by describing how this method works.

Note first that `mutation_class_iter` is, as the name already indicates, an *iterator*. This means that the next element is only computed if the iterator is asked to do so. Here is an example. One might be interested if there exists a seed or quiver in a given infinite mutation class having a certain property. Of course, we cannot test all elements, but we can construct the iterator and then let the computer run through the elements, constructing one after the other, and checking this property. If the program finds an element having the property, one could halt the process and return the element, together with all mutations applied to the initial element. If the computer keeps running, you might (or might not) get convinced that such an element does not exist.

The command `mutation_class_iter` has five (resp. six) optional arguments if it is acting on a cluster seed (resp. quiver). The additional optional argument for quivers is `data_type` which is initially set to `'quiver'` but can also be allowed to be `matrix`, `digraph`, `dig6`, or `path`. This argument does not appear in the cluster seed since the data type is assumed to be a cluster seed here.

The second optional argument is `depth`, which is set to be 'infinity' by default and instructs how large a "ball" in the mutation-equivalence-class around the initial input is supposed to be constructed. If the cluster algebra is of finite type (resp. finite mutation type) however then a depth of infinity will eventually construct the entire mutation class, when the original input is a cluster seed (resp. quiver).

Another optional argument is `show_depth`, which allows the user to print extra information of the actual depth, the number of constructed seeds or quivers, and the elapsed time. It is set to be `False` by default. The argument `up_to_equivalence` works differently depending on whether the input is a cluster seed or a quiver. In the default case `True`, cluster seeds are considered up to simultaneous row and column permutations and quivers are considered unlabeled; see Remark 3.4. Otherwise, equivalence of seeds and quivers are not considered.

```
sage:  S = ClusterSeed(['A',2]);
sage:  S.cluster_class()
```

$$\left[[x_0, x_1], \left[x_0, \frac{x_0+1}{x_1}\right], \left[\frac{x_1+1}{x_0}, x_1\right], \left[\frac{x_0+x_1+1}{x_0x_1}, \frac{x_0+1}{x_1}\right], \left[\frac{x_1+1}{x_0}, \frac{x_0+x_1+1}{x_0x_1}\right]\right]$$

```
sage:  S.cluster_class(up_to_equivalence=False)
```

$$\left[[x_0, x_1], \left[x_0, \frac{x_0+1}{x_1}\right], \left[\frac{x_1+1}{x_0}, x_1\right], \left[\frac{x_1+1}{x_0}, \frac{x_0+x_1+1}{x_0x_1}\right], \left[\frac{x_0+x_1+1}{x_0x_1}, \frac{x_0+1}{x_1}\right],\right.$$

$$\left.\left[\frac{x_0+x_1+1}{x_0x_1}, \frac{x_1+1}{x_0}\right], \left[\frac{x_0+1}{x_1}, \frac{x_0+x_1+1}{x_0x_1}\right], \left[x_1, \frac{x_1+1}{x_0}\right], \left[\frac{x_0+1}{x_1}, x_0\right], [x_1, x_0]\right]$$

The argument `sink_source` is set to be `False` by default, but if set to `True`, then only mutations at sinks and sources are performed. This option is helpful for working with bipartite seeds or studying the BGP reflection functors on quiver representations.

Finally, the last argument `return_paths`, again `False` by default, will keep track of the shortest mutation sequence that can be used to produce a given seed (or quiver) from the initial one. This data can be accessed by other commands and then utilized for future work. Note that such a sequence is not unique so accessing this shortest sequence during different computational sessions might not give the same result but for most purposes a single example of the mutation sequence between two seeds is sufficient data.

With this iterator, one can then call `mutation_class` which will output the associated list of seeds or quivers in the mutation class. However, since this output cannot be infinite, the argument `depth` cannot be `infinity` unless the input is of finite (resp. finite mutation type). The data associated to the optional arguments is also returned at this time. The commands `b_matrix_class` and `cluster_class`, which each can only be performed on a cluster seed, work analogously. The algorithm for `variable_class`, which again only works on a cluster seed, requires a little more explanation.

The procedure for `variable_class_iter` starts by running through an iterator for the mutation class and by yielding all found cluster variables. However, since the set of cluster variables is dwarfed by the number of clusters, this search-based algorithm is quite slow.

On the other hand, if we are in the lucky situation that the initial cluster is bipartite, then we can use [FZ07, Theorem 8.8] to efficiently compute the variable class.

Theorem 4.16 (Theorem 8.8 of [FZ07])**.** *Suppose that an exchange matrix B is bipartite, and its Cartan counterpart $A = A(B)$ is indecomposable.*

1) If A is of finite type, then the corresponding bipartite belt (see Definition 4.17) has the following periodicity property: the labeled seeds Σ_m and $\Sigma_{m+2(h+2)}$ are equal to each other for all $m \in \mathbb{Z}$. Here, h is the Coxeter number of the corresponding Cartan matrix A.

2) If A is of infinite type, then all of the elements $x_{i;2m}$, denoting the n cluster variables of $\Sigma_{2m} = (\{x_{1;2m}, x_{2;2m}, \ldots, x_{n;2m}\}, B)$ as m ranges over the integers are distinct Laurent polynomials in the initial data.

Note that in this theorem, the *Cartan counterpart* of B (see Section 5) is the (generalized) *Cartan matrix* $A = A(B) = (a_{ij})$ defined by

$$a_{ij} = \begin{cases} 2 & \text{if } i = j \\ -|b_{ij}| & \text{if } i \neq j \end{cases}.$$

Definition 4.17. We use $\Sigma_0 = (\mathbf{x_0}, B)$ to denote an initial bipartite seed and let μ_+ (resp. μ_-) denote the concatenation of all mutations at sources (sinks) of the quiver $Q(B)$[7]. Observe that $\mu_+(B) = \mu_-(B) = -B$.

[7]Since sources and sinks are not adjacent, the factors of μ_+ (resp. μ_-) commute with one another, hence why μ_+ and μ_- are well-defined.

Define the associated *bipartite belt* to be the seeds $\Sigma_m = (\mathbf{x_m}, (-1)^m B)$ for $m \in \mathbb{Z}$, defined recursively by

$$\Sigma_r = \begin{cases} \mu_+(\Sigma_{r-1}) & \text{if } r \text{ is odd} \\ \mu_-(\Sigma_{r-1}) & \text{if } r \text{ is even.} \end{cases}$$

As a consequence, given an initial bipartite seed $(\mathbf{x}, B)$, it is sufficient to mutate all vertices labeling sinks in $Q(B)$ followed by mutating all vertices labeling sources in $Q(B)$, and iterate. We will get no repeats in this list and thus the most efficient way to obtain all cluster variables in the case of a finite type cluster algebra[8].

Our algorithm thus first checks if the initial seed is bipartite for this reason. If not, it proceeds as above trying to mutate in all directions.

It is a difficult computational problem to find a mutation sequence, if one exists, from an initial non-bipartite seed to a bipartite one, so it is not computationally feasible to use the shortcut if we do not have a bipartite seed at hand. However, since our proceeding is doing a search through all seeds mutation-equivalent to the initial one anyway as its default behavior if we get lucky and find a bipartite seed, the program can record this path and take advantage of this find.

In the case that the search algorithm finds a bipartite seed, the algorithm then does the following procedure instead:

1) Starts over at the initial seed.

2) Mutates along the recorded path to get to the bipartite seed Σ_0.

3) Mutate along the bipartite belt the appropriate distance from there in both directions (i.e. applying μ_+ first or μ_- first).

In step (3) the appropriate distance is either the period $2(h+2)$ in the case of a cluster algebra of finite type or the `depth` chosen beforehand by the user. Note well that the meaning of `depth` is actually different here, as the algorithm will no longer spread out in all directions. Instead, the argument `depth` now instructs the computer how many iterations of the bipartite belt to use. The program will actually output the cluster variables found on the way to the bipartite seed Σ_0, as well as all cluster variables in the seeds $\{\Sigma_m : m \in \mathbb{Z}, |m| \leq \texttt{depth}\}$.

Since in the case of infinite type, not all cluster variables can be reached by using the bipartite belt, for example even cluster variables lying in clusters two mutations away from the bipartite seed might not be reachable (see the bipartite $\tilde{A}_{2,2}$ example below), the optional argument `ignore_bipartite_belt=False` is included. If set to be `True`, the original (albeit slower) algorithm of mutating in all directions out to a certain depth is utilized even if a bipartite seed is found.

```
sage:  S = ClusterSeed(['A',[2,2],1]); S.b_matrix(); S.is_bipartite()
```

$$\begin{pmatrix} 0 & -1 & 0 & -1 \\ 1 & 0 & 1 & 0 \\ 0 & -1 & 0 & -1 \\ 1 & 0 & 1 & 0 \end{pmatrix} \qquad \texttt{True}$$

```
sage:  S.variable_class(depth=1)
    Found a bipartite seed -
          constructing the variable class into its bipartite belt.
```

[8] If the cluster algebra is of infinite type, one can also mutate along the bipartite belt to efficiently generate a large list of cluster variables but *not all cluster variables are reachable* in this way.

$$\left[x_0, x_1, x_2, x_3, \frac{x_1x_3+1}{x_0}, \frac{x_0x_2+1}{x_1}, \frac{x_1x_3+1}{x_2}, \frac{x_0x_2+1}{x_3}, \frac{x_1^2x_3^2+x_0x_2+2x_1x_3+1}{x_0x_1x_2}, \frac{x_0^2x_2^2+2x_0x_2+x_1x_3+1}{x_0x_1x_3}, \frac{x_1^2x_3^2+x_0x_2+2x_1x_3+1}{x_0x_2x_3}, \frac{x_0^2x_2^2+2x_0x_2+x_1x_3+1}{x_1x_2x_3}\right]$$

If we look at the output from `S.variable_class(depth=2)` or higher depth, we will see that the denominators grow larger and larger but no denominator of x_0x_1 appears. Compare this output with the examples below.

```
sage: S.mutate([0,1]); S.cluster()
```

$$\left[\frac{x_1x_3+1}{x_0}, \frac{x_0x_2+x_1x_3+1}{x_0x_1}, x_2, x_3\right]$$

```
sage: S.variable_class(depth=2, ignore_bipartite_belt=True)
```

$$\left[x_0, x_1, x_2, x_3, \frac{x_1x_3+1}{x_0}, \frac{x_0x_2+1}{x_1}, \frac{x_1x_3+1}{x_2}, \frac{x_0x_2+x_1x_3+1}{x_0x_1}, \frac{x_0x_2+x_1x_3+1}{x_0x_3}, \frac{x_0x_2+x_1x_3+1}{x_1x_2}, \frac{x_1^2x_3^2+x_0x_2+2x_1x_3+1}{x_0x_1x_2}, \frac{x_0^2x_2^2+2x_0x_2+x_1x_3+1}{x_0x_1x_3}, \frac{x_1^3x_3^3+x_0^2x_2^2+2x_0x_1x_2x_3+3x_1^2x_3^2+2x_0x_2+3x_1x_3+1}{x_0^2x_1x_2x_3}\right]$$

5. Associahedra and the cluster complex

Before looking at associahedra, the cluster complex and their implementations, we need to start with some basic background on root systems for (generalized) Cartan matrices. For further details, we refer to [Hum72, Kac94].

Definition 5.1 (Generalized Cartan matrix)**.** An $n \times n$-matrix $A = (a_{ij})$ with integer entries is called a *generalized Cartan matrix* if

- $a_{ii} = 2$,
- $a_{ij} < 0$ for $i \neq j$,
- A is symmetrizable, i.e., there exists a diagonal matrix D with positive entries such that DA is symmetric.

A generalized Cartan matrix is called *of finite type* if DA is positive definite, and *of affine type* if DA is positive semi-definite.

Recalling the definition of B-matrices, we see that we can associate a generalized Cartan matrix to every B-matrix (see [FZ03b, (1.6)]). The terms *finite* and *affine* come from their connections to finite and affine *Lie algebras*. Indecomposable generalized Cartan matrices of finite type (resp. of affine type) classify Lie algebras of finite type (resp. of affine type).

A *realization* of a Cartan matrix A (of finite type) is a (rational, real, or complex) vector space V with distinguished basis $\Delta = \{\alpha_i : 0 \leq i < n\}$, and with dual space V^* with distinguished basis $\Delta^\vee = \{\alpha_i^\vee : 0 \leq i < n\}$, together with the pairing $\langle \alpha_i^\vee, \alpha_j \rangle = a_{ij}$. For $\beta \in V$ (resp. $\beta^\vee \in V^\vee$), we write $[\beta, \alpha_i]$ (resp. $[\beta^\vee, \alpha_i^\vee]$) for the coefficient of α_i in β (resp. $\alpha_i^\vee$ in $\beta^\vee$).

Define a *reflection* on V by

$$s_i(\alpha_j) = \alpha_j - a_{ij}\alpha_i,$$

and define moreover, the *Weyl group* by $W = \langle s_i : 0 \leq i < n\rangle \leq \mathrm{Aut}(V)$ and the *root system* by

$$\Phi = \big\{\omega(\alpha) : \omega \in W, \alpha \in \Delta\big\}.$$

It can be shown that Φ can be written as $\Phi^+ \cup \Phi^-$ where

$$\Phi^+ = \{\beta \in \Phi : [\beta, \alpha] > 0 \text{ for all } \alpha \in \Delta\},$$

and $\Phi^- = \{-\beta : \beta \in \Phi^+$. The elements in Φ are called *roots*, the elements in Φ^+ are called *positive roots*, and the elements in Δ are called *simple roots*.

Theorem 5.2 (Theorem 1.9 of [FZ03b]). *Let $\mathcal{A}$ be a Cluster algebra of finite type and let $\Phi_{\geq -1} = \Phi^+ \cup -\Delta$ be the set of* almost positive roots *of the root system of the associated Cartan type given by the positive roots together with the simple negative roots. There exists a unique bijection between almost positive roots and the cluster variables for $\mathcal{A}$ for which the simple negative root $-\alpha$ is mapped to x_α and, for positive roots,*

$$\sum_{\alpha\in\Delta} n_\alpha \alpha \mapsto \frac{P_\alpha}{\prod x_\alpha^{n_\alpha}},$$

with P_α having nonzero constant term. Here, x_{α_i} stands for x_i for an appropriate ordering $\Delta = \{\alpha_0, \ldots, \alpha_{n-1}\}$.

This connection in the finite types can be used in the cluster algebra package as follows:

```
sage: for f in ClusterSeed(['A',2]).variable_class():
....: print f, f.almost_positive_root()
```

$$\begin{array}{rl} x_0 & -\alpha_1 \\ x_1 & -\alpha_2 \\ (x_1+1)/x_0 & \alpha_1 \\ (x_0+1)/x_1 & \alpha_2 \\ (x_0+x_1+1)/(x_0x_1) & \alpha_1+\alpha_2 \end{array}$$

```
sage: f
```

$$(x_0 + x_1 + 1)/(x_0x_1)$$

```
sage: root = f.almost_positive_root(); root
```

$$\alpha_1 + \alpha_2$$

```
sage: root.parent()
      Root lattice of the Root system of type ['A', 2]
```

5.1. **Generalized associahedra.** In this section, we will define generalized associahedra and describe how they can be realized as polytopal complexes in finite types. We will see then how these polytopal complexes are implemented in `sage`. Generalized associahedra beyond finite type are not yet feasible as the needed tools to deal with infinite types are not yet developed. We start with the definition of generalized associahedra (not necessarily of finite type).

Definition 5.3 (Generalized associahedron)**.** The *generalized associahedron* associated to a cluster algebra $\mathcal{A}$ can be defined as the exchange graph of the mutation class of all cluster seeds for $\mathcal{A}$. This is the unoriented graph with vertices given by the set of all cluster seeds, and with an edge joining two clusters if they can be obtained from each other by a mutation.

Generalized associahedra reduce in classical types to known constructions, see e.g. [FZ03b, Section 12]. By [FZ03b, Theorem 1.12], a cluster seed of finite type is uniquely determined by its cluster, and two seeds are obtained from each other by a mutation if and only their clusters differ by exactly one cluster variable, see Theorem 4.8. In finite types, there exist realizations as polytopal complexes, see [CFZ02]. Let S_+, S_- be the bipartition of the simple reflections $S = \{s_\alpha : \alpha \in \Delta\}$ corresponding to the simple roots in Δ. This means that S_+ and S_- are chosen in such a way that the reflections in each pairwise commute. Observe that the fact that all quivers of finite type are bipartite ensures that such bipartitions always exist. Define two piecewise linear operators τ_+ and τ_- on V by

$$\tau_\epsilon(\beta) = \begin{cases} \beta & \text{if } \beta = -\alpha \text{ for } s_\alpha \in S_{-\epsilon} \\ \prod_{s\in S_\epsilon} s(\beta) & \text{otherwise,} \end{cases}$$

and let

$$\rho^\vee = \frac{1}{2}\sum_{\beta\in\Phi^+} \beta^\vee \in V^*.$$

In [CFZ02, Theorem 1.1], it is shown that every $\langle\tau_+,\tau_-\rangle$-orbit in $\Phi_{\geq -1}$ intersects $-\Delta$. Moreover, $\alpha_i, \alpha_j \in -\Delta$ lie in the same orbit if and only if $\alpha_i = -\omega_0(\alpha_j)$ where ω_0 is the (unique) *longest element* in W. Thus, the coefficients $[\rho^\vee, \alpha_i^\vee]$ and $[\rho^\vee, \alpha_j^\vee]$ coincide; for $\beta \in \Phi_{\geq -1}$, set c_β to be this coefficient. After identifying φ with the n-tuple $(\langle\varphi, \alpha_i\rangle)_{0\leq i<n}$, define the half-space

$$H^+(\beta) := \{\varphi \in \mathbb{R}^n : \langle\varphi,\beta\rangle \leq c_\beta\}$$

to obtain the polytopal realization of the generalized associahedron by

$$\mathrm{Ass}(\Phi) = \bigcap_{\beta\in\Phi^+} H^+(\beta) \subseteq \mathbb{R}^n.$$

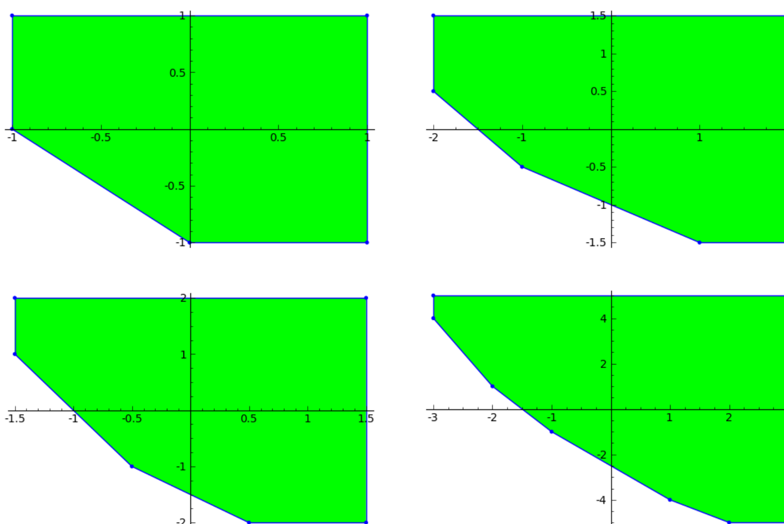
The operators τ_+ and τ_- are implemented in sage as operators for the root space.

```
sage: S = RootSystem(['A',2]).root_space()
sage: tau_plus, tau_minus = S.tau_plus_minus()
sage: for beta in S.almost_positive_roots():
....:     print beta, tau_plus(beta), tau_minus(beta)
....:     print
```

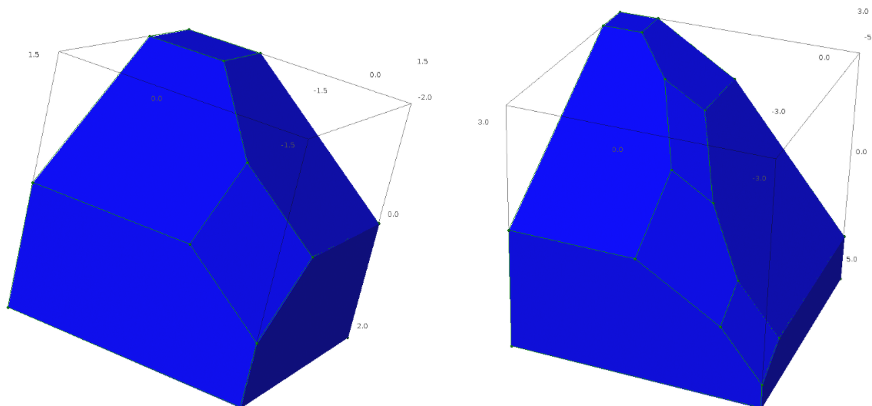
$$-\alpha_1, \quad \alpha_1, \quad -\alpha_1$$
$$\alpha_1, \quad -\alpha_1, \quad \alpha_1+\alpha_2$$
$$\alpha_1+\alpha_2, \quad \alpha_2, \quad \alpha_1$$
$$-\alpha_2, \quad -\alpha_2, \quad \alpha_2$$
$$\alpha_2, \quad \alpha_1+\alpha_2, \quad -\alpha_2$$

```
sage: AssoA2 = Associahedron(['A',2]); AssoA2
```

```
                 The generalized associahedron of type ['A', 2]
                                   having 2 dimensions and 5 vertices

sage:  AssoB2 = Associahedron(['B',2]); AssoB2

                 The generalized associahedron of type ['B', 2]
                                   having 2 dimensions and 6 vertices

sage:  AssoC2 = Associahedron(['C',2]); AssoC2

                 The generalized associahedron of type ['C', 2]
                                   having 2 dimensions and 6 vertices

sage:  AssoG2 = Associahedron(['G',2]); AssoG2

                 The generalized associahedron of type ['G', 2]
                                   having 2 dimensions and 8 vertices

sage:  AssoA2.show(); AssoB2.show(); AssoC2.show(); AssoG2.show()
```



```
sage:  AssoA3 = Associahedron(['A',3]); AssoA3

                 The generalized associahedron of type ['A', 3]
                                  having 3 dimensions and 14 vertices

sage:  AssoB3 = Associahedron(['B',3]); AssoB3

                 The generalized associahedron of type ['B', 3]
                                  having 3 dimensions and 20 vertices

sage:  AssoA3.show(); AssoB3.show()
```



The associahedron of type A_3 has 14 vertices (13 of which are visible, the 14th is the origin, which corresponds to the cluster $\{-\alpha_1, -\alpha_2, -\alpha_3\}$). As well, the 9 facets corresponds to the almost positive roots, where the hyperplane $x_i = c_{-\alpha_i}$ correspond to the simple negative root $-\alpha_i$. Every vertex corresponds to exactly 3 hyperplanes, and in type B_3, we have 20 vertices and 12 facets, as desired.

5.2. **The cluster complex.** As with associahedra, we will define the cluster complex in general and then discuss the implementation for finite types.

Definition 5.4 (Cluster complex)**.** The *cluster complex* associated to a cluster algebra $\mathcal{A}$ can be defined to be the simplicial complex with vertices being the cluster variables for $\mathcal{A}$ and with facets being the clusters.

As we have seen, cluster variables in finite types are in bijection with almost positive roots. We use this description in the implementation of the cluster complex.

```
sage:  ClusterComplex(['A',2])
           Simplicial complex with 5 vertices and 5 facets
sage:  ClusterComplex(['A',3])
           Simplicial complex with 9 vertices and 14 facets
sage:  Delta = ClusterComplex(['B',3]); Delta
           Simplicial complex with 12 vertices and 20 facets
```

In the following example, we see how we can use other `sage` packages to further study objects we work with. As the cluster complex is a simplicial complex, there now exists various possible methods. For example, we can compute its *homology*,

```
sage:  Delta.homology()
```

$$\{0 : 0, 1 : 0, 2 : \mathbb{Z}\}$$

This is as expected, as this simplicial complex is the boundary complex of a triangulated polytope, and thus shellable and Cohen-Macaulay.

6. Methods and attributes

In this section, we describe the different classes defined in this package, and list their attributes and methods. For the "key" methods, we also give descriptions of the algorithms.

In general, attribute names start with an underscore to emphasize that they should not be used directly but only through appropriate methods. As an example, a cluster seed has an attribute _M in which its exchange matrix is stored and a method b_matrix which is used to get the exchange matrix. The difference is that the method returns a copy of its exchange matrix, so it is safe to work with this matrix and to modify it without accidentally modifying the seed itself.

```
sage: S = ClusterSeed(['A',3]);
sage: M1 = S._M; M2 = S.b_matrix();
sage: M1 == M2
True
sage: M1 is M2
False
```

6.1. **Skew-symmetrizable matrices.** We briefly want to describe the algorithm used to determine whether a square matrix B is skew-symmetrizable, which also determines the associated diagonal matrix D in the affirmative case. It was written by F. Block, F. Saliola, and C. Stump during the sage days 20.5 at the Fields Institute, Toronto, Canada, in May 2010.

Algorithm 6.1. *Let $B = (b_{ij})_{1\leq i,j\leq n}$ be the input square matrix of dimension n, and let $D = (d_i)_{1\leq i\leq n}$ be the diagonal matrix with positive coefficients we want to construct. We use the equivalent description of skew-symmetrizablility given by the property*

$$d_i b_{ij} = -d_j b_{ji} \text{ for all } i, j.$$

(1) *Check if $b_{ii} = 0$ for all i. If this is not the case, return* False*,*
(2) *let k be the smallest integer such that d_k is not yet determined,*
(3) *set $d_k = 1$,*
(4) *for $i \in \{1, \ldots, n\}$ such that $b_{ik} \neq 0$ and d_i is not yet determined, do*
    (a) *set $d_i = -d_k b_{ki}/b_{ik}$,*
    (b) *if $d_i \leq 0$ return* False*.*
    (c) *if* any $\big(d_i b_{ij} \neq -d_j b_{ji}\big)$ *for j such that d_j is already determined, return* False*.*
(5) *repeat step (4) with k given by all integers for which d_i was set since we passed step (3) the last time,*
(6) *if D is not yet completely determined,* goto *step (2),*
(7) *return D.*

6.2. **QuiverMutationType.** For coding reasons, we distinguish between the classes QuiverMutationType_Irreducible and QuiverMutationType_Reducible, but we refer here to both as QuiverMutationType. Objects of those types are unique, i.e., there exists only one object of a given quiver mutation type.

```
sage: mut_type1 = QuiverMutationType('A',3)
sage: mut_type2 = QuiverMutationType('A',3)
sage: mut_type1 is mut_type2
True
```

All the data for quiver mutation types is hard-coded. In particular, this concerns the graphs and digraphs, and the class size.

To construct a quiver mutation type, the function `QuiverMutationType` is called. An irreducible quiver mutation type takes 3 parameters, the `letter`, the `rank` or `bi_rank`, and the `twist`, see the description below. Those calls are best explained in examples. Observe that the call arguments can be also wrapped into a list or tuple. We suppress the output whenever the output coincide with the input.

- finite types

```
sage: QuiverMutationType('A',1);
sage: QuiverMutationType('A',5);
sage: QuiverMutationType('B',2);
sage: QuiverMutationType('B',5);
sage: QuiverMutationType('C',2)
                                  ['B', 2]
sage: QuiverMutationType('C',5);
sage: QuiverMutationType('D',2)
                       [ ['A', 1], ['A', 1] ]
sage: QuiverMutationType('D',3)
                                  ['A', 3]
sage: QuiverMutationType('D',4);
sage: QuiverMutationType('E',6);
sage: QuiverMutationType('E',7);
sage: QuiverMutationType('E',8);
sage: QuiverMutationType('F',4);
sage: QuiverMutationType('G',2);
```

- affine types

```
sage: QuiverMutationType('A',(1,1),1);
sage: QuiverMutationType('A',(2,4),1);
sage: QuiverMutationType('BB',1,1)
                             ['A', [1, 1], 1]
sage: QuiverMutationType('BB',2,1);
sage: QuiverMutationType('BB',4,1);
sage: QuiverMutationType('CC',1,1)
                             ['A', [1, 1], 1]
sage: QuiverMutationType('CC',2,1);
sage: QuiverMutationType('CC',4,1);
sage: QuiverMutationType('BC',1,1);
sage: QuiverMutationType('BC',5,1);
sage: QuiverMutationType('BD',3,1);
sage: QuiverMutationType('BD',5,1);
sage: QuiverMutationType('CD',3,1);
sage: QuiverMutationType('CD',5,1);
sage: QuiverMutationType('D',4,1);
sage: QuiverMutationType('D',6,1);
sage: QuiverMutationType('E',6,1);
sage: QuiverMutationType('E',7,1);
sage: QuiverMutationType('E',8,1);
sage: QuiverMutationType('F',4,1);
sage: QuiverMutationType('F',4,-1);
sage: QuiverMutationType('G',2,1);
```

```
sage: QuiverMutationType('G',2,-1);
```

- hyperbolic types

```
sage: QuiverMutationType('E',6,[1,1]);
sage: QuiverMutationType('E',7,[1,1]);
sage: QuiverMutationType('E',8,[1,1]);
```

- mutation-finite types
  - rank 2

```
sage: QuiverMutationType('R2',(1,1),2)
                                   ['A', 2]
sage: QuiverMutationType('R2',(1,2),2)
                                   ['B', 2]
sage: QuiverMutationType('R2',(1,3),2)
                                   ['G', 2]
sage: QuiverMutationType('R2',(1,4),2)
                                ['BC', 1, 1]
sage: QuiverMutationType('R2',(1,5),2);
sage: QuiverMutationType('R2',(2,2),2)
                             ['A', [1, 1], 1]
sage: QuiverMutationType('R2',(3,5),2);
```

  - exceptional types

```
sage: QuiverMutationType('V',4,2);
sage: QuiverMutationType('W',4,2);
sage: QuiverMutationType('W',4,-2);
sage: QuiverMutationType('X',6,2);
sage: QuiverMutationType('X',7,2):
sage: QuiverMutationType('Y',6,2);
sage: QuiverMutationType('Z',6,2);
sage: QuiverMutationType('Z',6,-2);
```

- mutation-infinite types
  - infinite type E

```
sage: QuiverMutationType('E',9,3)
                                 ['E', 8, 1]
sage: QuiverMutationType('E',10,3);
sage: QuiverMutationType('E',12,3);
sage: QuiverMutationType('AE',(1,1),3);
sage: QuiverMutationType('AE',(1,4),3);
sage: QuiverMutationType('BE',5,3);
sage: QuiverMutationType('CE',5,3);
sage: QuiverMutationType('DE',6,3);
```

  - Grassmannian types – the second parameter (a, b) must satisfy $1 \leq a < b$ and one obtains a grid graph of width $a-1$ and height $b-a-1$

```
sage: QuiverMutationType('GR',(2,4),3)
                                   ['A', 1]
sage: QuiverMutationType('GR',(2,6),3)
                                   ['A', 3]
sage: QuiverMutationType('GR',(3,6),3)
                                   ['D', 4]
```

```
sage: QuiverMutationType('GR',(3,7),3)
                              ['E', 6]
sage: QuiverMutationType('GR',(3,8),3)
                              ['E', 8]
sage: QuiverMutationType('GR',(3,9),3)
                         ['E', 8, [1,1]]
sage: QuiverMutationType('GR',(3,10),3);
```

– triangular types – the second parameter gives the size of the graph

```
sage: QuiverMutationType('TR',2,3)
                              ['A', 3]
sage: QuiverMutationType('TR',3,3)
                              ['D', 6]
sage: QuiverMutationType('TR',4,3)
                         ['E', 8, [1, 1]]
sage: QuiverMutationType('TR',5,3);
```

– type T – the second parameter gives the lengths of the three legs

```
sage: QuiverMutationType('T',(1,1,1),3)
                              ['A', 1]
sage: QuiverMutationType('T',(1,1,4),3)
                              ['A', 4]
sage: QuiverMutationType('T',(1,4,4),3)
                              ['A', 7]
sage: QuiverMutationType('T',(2,2,2),3)
                              ['D', 4]
sage: QuiverMutationType('T',(2,2,4),3)
                              ['D', 6]
sage: QuiverMutationType('T',(2,3,3),3)
                              ['E', 6]
sage: QuiverMutationType('T',(2,3,4),3)
                              ['E', 7]
sage: QuiverMutationType('T',(2,3,5),3)
                              ['E', 8]
sage: QuiverMutationType('T',(2,3,6),3)
                            ['E', 8, 1]
sage: QuiverMutationType('T',(2,3,7),3)
                           ['E', 10, 3]
sage: QuiverMutationType('T',(3,3,3),3)
                            ['E', 6, 1]
sage: QuiverMutationType('T',(3,3,4),3);
```

- reducible types

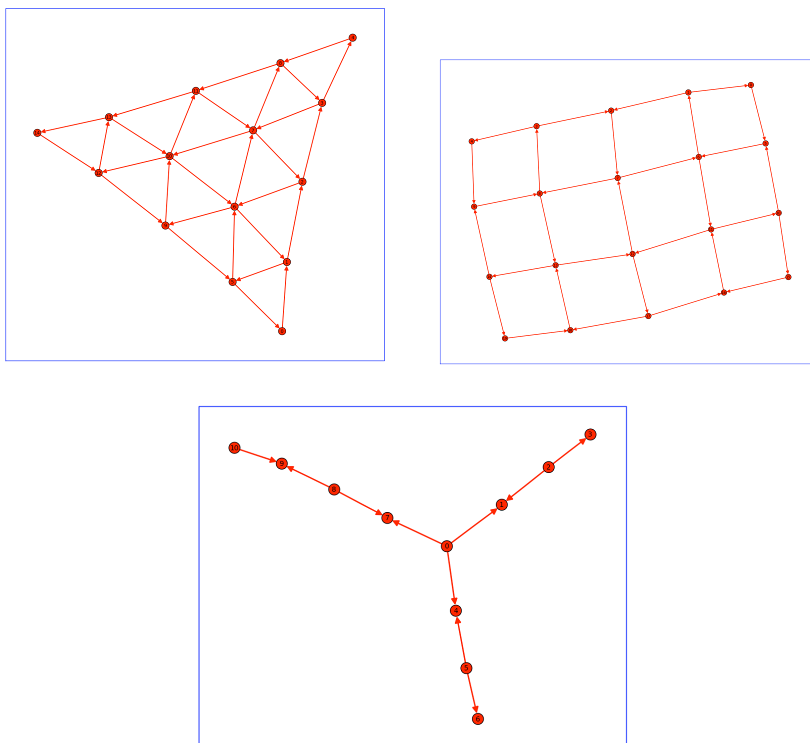
```
sage: QuiverMutationType(['A',3],['B',4])
                   [ ['A', 3], ['B', 4] ]
```

As described in Section 4.2, one can use also Kac's classification types [Kac94].

Remark 6.2. Most of the above types have already been explained as Dynkin diagrams, appear in Kac's list, or in the classification work of Derksen-Owen [DO08], and Felikson-Shapiro-Tumarkin [FST08, FST10]. The exceptions to these are the triangular seeds, Grassmannian seeds, and the "T" seeds. The first two of these describe a certain family of quivers that have certain shapes (as triangles and grids,

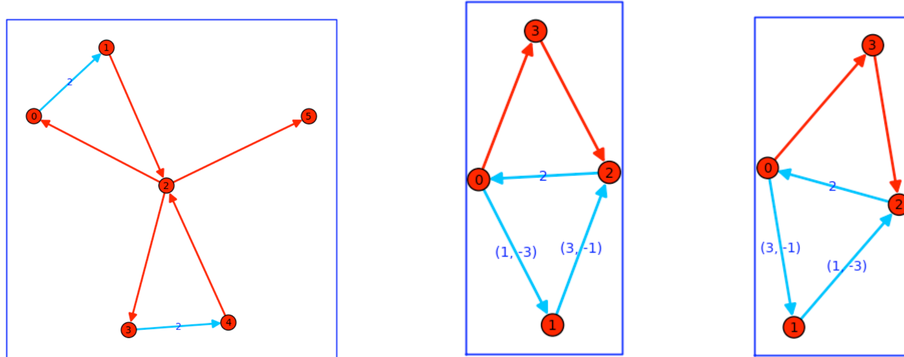
respectively) and correspond to certain coordinate rings of geometric objects. (See Examples 4.4 and 4.6 of [Kel2] or the source papers [BFZ05] and [Sco06].) The "T" family are those which correspond to "Dynkin diagrams" of the shape of a T with a certain number of vertices on each arm and one central vertex.

```
sage:  ClusterSeed(['TR',5,3]).show()
sage:  ClusterSeed(['GR',[5,11],3]).show()
sage:  ClusterSeed(['T',[4,4,5],3]).show()
```



We also illustrate a self-dual and two dual non-simply laced exceptional mutation-finite cases here too.

```
sage:  ClusterSeed(['X',6,2]).show()
sage:  S = ClusterSeed(['W',4,2]); S.show()
sage:  S = ClusterSeed(['W',4,-2]); S.show()
```



The attributes of `QuiverMutationType` are given by

- `_letter`
  The string containing the letter(s) of the classification type.

- `_rank`
  The number of vertices in the standard quiver.

- `_bi_rank`
  Is `None` except for affine type A, where it denotes $[a, b]$ with $a+b$ being the rank and $a \leq b$ are the number of edges in the acyclic orientation of the standard quiver.

- `_twist`
  Depends on the type of the classification type, and can be one of the following:
  - `None` for finite types,
  - `1` for affine types,
  - `[1,1]` for elliptic types,
  - `2` for finite mutation types which are not finite or elliptic,
  - `3` for infinite mutation types.

- `_graph`
  Graph representing the underlying graph of the standard quiver.

- `_digraph`
  Digraph representing the underlying graph of the standard quiver.

- `_description`
  The string representation of the mutation class.

- `_info`
  Dictionary containing the keys
  - irreducible,
  - finite,
  - affine,

  - elliptic,
  - simply_laced,
  - mutation_finite, and
  - irreducible_components.

  The values are `True` or `False`, except for `irreducible_components` which is a list containing the irreducible components.

The methods of `QuiverMutationType` are given by

- `__eq__(self,other)`
  Returns `True`, iff self and other represent the same quiver mutation type. As quiver mutation types are unique (i.e., there exists at most one object representing a given quiver mutation type), this method simply returns `self is other`.

- `_repr_(self)`
  Returns the string representation of self.

- `plot(self, circular=False, directed=True)`
  Returns a random or circular, directed or undirected plot of self.

- `show(self, circular=False, directed=True)`
  Shows the plot of self.

- `rank(self)`
  Returns the rank (i.e., the number of vertices) of self.

- `coxeter_diagram(self)`
  Returns the Coxeter diagram of self

```
sage: QuiverMutationType(['A',5]).coxeter_diagram()
                    Coxeter diagram of rank 5
sage: QuiverMutationType(['A',3],['B',3]).coxeter_diagram()
                    Coxeter diagram of rank 8
```

- `b_matrix(self)`
  Returns the exchange matrix of self

```
sage: QuiverMutationType(['A',5]).b_matrix()
```

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ -1 & 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 & -1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

```
sage: QuiverMutationType(['A',3],['B',3]).b_matrix()
```

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 & 0 & -1 \\ 0 & 0 & 0 & 0 & 2 & 0 \end{pmatrix}$$

- `standard_quiver(self)`
  Returns the standard quiver of self.

- `cartan_matrix(self)`
  Returns the Cartan matrix of self which is obtained from its exchange matrix by replacing the positive entries by negative, and replace the 0's on the main diagonal by 2's.

```
sage: QuiverMutationType('A',5).cartan_matrix()
```

$$\begin{pmatrix} 2 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 2 \end{pmatrix}$$

```
sage: QuiverMutationType(['A',3],['B',3]).cartan_matrix()
```

$$\begin{pmatrix} 2 & -1 & 0 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 & 0 \\ 0 & -1 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & -1 & 0 \\ 0 & 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & 0 & -2 & 2 \end{pmatrix}$$

- `class_size(self)`
  Returns the number of quivers which are mutation-equivalent to self, up to isomorphism (**Warning:** several class sizes are only conjectured, see Section 4.3).

```
sage: QuiverMutationType(['A',22],['BD',16,1]).class_size()
                    4257164518523691840
sage: QuiverMutationType(['GR',[4,9],3]).class_size()
```

$$\infty$$

- `dual(self)`
  Returns the dual quiver mutation type of self.

```
sage: QuiverMutationType('A',4).dual()
                         ['A', 4]
sage: QuiverMutationType('B',4).dual()
                         ['C', 4]
```

- `is_irreducible(self)`
  Returns `True`, iff self is irreducible.

```
sage: QuiverMutationType('A',4).is_irreducible()
                           True
sage: QuiverMutationType(['A',3],['B',3]).is_irreducible()
                           False
```

- `is_mutation_finite(self)`
  Returns `True`, iff self is of finite mutation type.

```
sage: QuiverMutationType(['GR',[4,8],3]).is_mutation_finite()
                           True
sage: QuiverMutationType(['GR',[4,9],3]).is_mutation_finite()
```

```
False
```

- `is_simply_laced(self)`
  Returns `True`, iff self is simply-laced.

- `is_finite(self)`
  Returns `True`, iff self is of finite type.

- `is_affine(self)`
  Returns `True`, iff self is of affine type.

- `is_elliptic(self)`
  Returns `True`, iff self is of elliptic type.

- `irreducible_components(self)`
  Returns a tuple containing the irreducible components of self.

```
sage: QuiverMutationType('A',5).irreducible_components()
(['A', 4],)
sage: QuiverMutationType(['A',3],['B',3]).irreducible_components()
(['A', 3], ['B', 3])
```

- `properties(self)`
  Prints all properties of self. See Section 4 for examples.

6.3. **Quiver.** The next class we want to describe it the class `Quiver`. It allows numerous ways to construct a quiver, several examples were described in Section 3.

- `QuiverMutationType`
- `list` or `tuple` representing a quiver mutation type
- `ClusterSeed`
- `matrix`: a skew-symmetrizable matrix which represents the exchange matrix
- `Quiver`
- `DiGraph`: the digraph must represent a quiver
- `list` of `tuples` representing the edge list of a digraph for a quiver

The attributes of `Quiver` are given by

- `_M`
  The exchange matrix of self.

- `_n`
  The number of cluster variables (which is the number of columns in the exchange matrix).

- `_m`
  The number of frozen variables (which is the number of rows minus the number of columns in the exchange matrix).

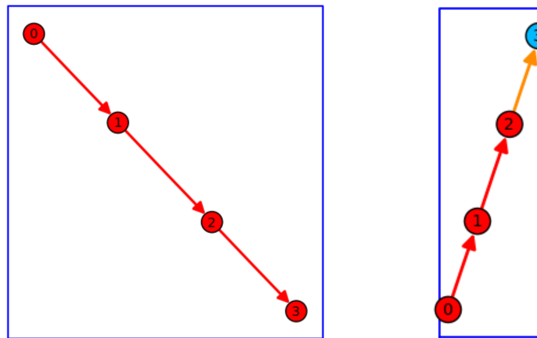
- `_description`
  The string representation of self.

- `_mutation_type`
  The mutation type of self, if known, `None` otherwise.

The methods of `Quiver` are given by

- `__init__(self, data, frozen=0)`
  Frozen sets the later vertices to be frozen

```
sage: Q1 = Quiver([(0,1),(1,2),(2,3)]); Q1
                    Quiver on 4 vertices
sage: Q2 = Quiver([(0,1),(1,2),(2,3)],frozen=1); Q2
          Quiver on 4 vertices with 1 frozen vertex
sage: Q1.show()
sage: Q2.show()
```



- `__eq__(self, other)`
  Returns `True`, iff the b-matrices of self and other coincide

```
sage: Q = Quiver(['A',5])
sage: T = Q.mutate( 2, inplace=False )
sage: Q.__eq__( T )
                         False
sage: T.mutate( 2 )
sage: Q.__eq__( T )
                         True
```

- `_repr_(self)`
  Returns the string representation of self

```
sage: Q = Quiver(['A',5])
sage: Q._repr_()
          "Quiver on 5 vertices of type ['A', 5]"
```

- `plot(self, circular=False, directed=True, mark=None)`
  Returns a random/circular and directed/undirected plot of self with a given vertex marked.

- `show(self, fig_size=1, circular=False, directed=True, mark=None)`
  Shows the plot of self.

- `interact(self, fig_size=1, circular=True)`
  Starts an interactive mode, as shown in Figure 1 at the end of Section 3.

- `save_image(self, filename, circular=False)`
  Saves the plot of self to filename.

- `b_matrix(self)`
  Returns the exchange matrix of self

```
sage: Quiver(['A',4]).b_matrix()
```

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ -1 & 0 & -1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

```
sage: Quiver(['B',4]).b_matrix()
```

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ -1 & 0 & -1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & -2 & 0 \end{pmatrix}$$

```
sage: Quiver(['D',4]).b_matrix()
```

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ -1 & 0 & -1 & -1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

```
sage: Quiver(QuiverMutationType([['A',2],['B',2]])).b_matrix()
```

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & -2 & 0 \end{pmatrix}$$

- `digraph(self)`
  Returns the underlying digraph of self

```
sage: Quiver(['A',4]).digraph()
                    Digraph on 4 vertices
```

- `n(self)`
  Returns the number of free vertices of self

```
sage: Q = Quiver([(0,1),(1,2),(2,3)],frozen=1)
sage: Q.n()
                    3
```

- `m(self)`
  Returns the number of frozen vertices of self

```
sage: Q = Quiver([(0,1),(1,2),(2,3)],frozen=1)
sage: Q.m()
                    1
```

- `canonical_label(self, certify=False)`
Returns an isomorphic quiver with canonical vertex labeling. This is based on the canonical labeling of digraphs using the corresponding method for digraphs by R.L. Miller based on [McK81]. If certify is `True`, a dictionary of the relabeling is also returned

```
sage:  Quiver(['A',4]).canonical_label(certify=True)
  (Quiver on 4 vertices of type ['A', 4], {0:0, 1:3, 2:1, 3:2})
```

- `is_acyclic(self)`
Returns `True`, iff self is acyclic.

- `is_bipartite(self, return_bipartition=False)`
Returns `True`, iff self is bipartite, if return_bipartition is `True`, the bipartition is returned

```
sage:  Quiver(['A',4]).is_bipartite(return_bipartition=True)
                    (set([0, 2]), set([1, 3]))
```

- `principal_restriction(self)`
Returns the principal restricting of self. This is obtained from self by deleting all frozen variables.

- `principal_extension(self)`
Returns the principal extension of self. This can be used only for seeds without frozen variables. Returns a new seed with exchange matrix of size $2n \times n$ given by the exchange matrix of self of size $n \times n$ with an additional identity matrix added below.

- `mutate(self, data, inplace=True)`
Mutates at a vertex or at a list of vertices, if inplace is `True`, self is modified, otherwise a new quiver is returned.

- `mutation_sequence(self, sequence, show_sequence=False, fig_size=1.2)`
Returns a list of quivers obtained from a sequence of mutations. If the parameter show_sequence is `True`, the sequence is shown with a given fig_size.

- `reorient(self,data)`
Reorients self with respect to the given total order, or with respect to an iterator of edges in self to be reverted.

  Warning: This often will change the mutation class of the quiver except if the quiver is a tree (see Theorem 4.7).

- `mutation_class_iter(self, depth=infinity, show_depth=False, return_paths=False, data_type='quiver', up_to_equivalence=True, only_sink_source=False)`
Returns an iterator which goes through the mutation class of self depending on several parameters

- `depth`: integer, only quivers with distance at most depth from self are returned
- `show_depth`: if `True`, the actual depth of the mutation is shown
- `return_paths`: if `True`, a shortest path of mutation sequences from self to the given quiver is returned as well
- `data_type`: can be one of the following:

  `quiver, matrix, digraph, dig6, path`

- `up_to_equivalence`: if True, only quivers up to equivalence are considered
- `sink_source`: if True, only mutations at sinks and sources are applied

- `mutation_class(self, depth=infinity, show_depth=False, return_paths=False, data_type='quiver', up_to_equivalence=True, only_sink_source=False)`
  Returns a list of all quivers in the corresponding iterator.

- `group_of_mutations(self)`
  Returns the group of mutations of self. **Warning:** The permutation group is very big! This group differs for quivers and for cluster seeds, as different cluster seeds may have the same exchange matrix and thus the same quiver. This group is defined to be the group of permutation given as follows. The ground set is the mutation class of self without taking equivalence of quivers into account, and the group is generated by the n involutions on this set given by mutation at the n different vertices. Observe that the analogous operation on the mutation class up to equivalence does not give a group (this can be easily checked in type A_3). Basically nothing is known about this group.

```
sage:  Q = Quiver(['A',2])
sage:  Q.group_of_mutations()
            Permutation Group with generators [(1,2)]
sage:  Q = Quiver(['A',3])
sage:  Q.group_of_mutations()
              Permutation Group with generators
           [(1,2)(3,4)(5,9)(6,7)(8,12)(10,11)(13,14),
            (1,3)(2,5)(4,6)(7,14)(8,11)(9,13)(10,12),
            (1,4)(2,3)(5,10)(6,8)(7,13)(9,14)(11,12)]
sage:  Q = Quiver(['B',2])
sage:  Q.group_of_mutations()
            Permutation Group with generators [(1,2)]
sage:  Q = Quiver(['B',3])
sage:  Q.group_of_mutations()
 Permutation Group with generators [(1,2)(3,4)(5,6)(7,10)(8,9),
    (1,3)(2,6)(4,5)(7,9)(8,10), (1,4)(2,3)(5,7)(6,8)(9,10)]
sage:  Q = Quiver(['A',1])
sage:  Q.group_of_mutations().cardinality()
                                 1
sage:  Q = Quiver(['A',2])
```

```
sage:  Q.group_of_mutations().cardinality()
                                  2
sage:  Q = Quiver(['A',3])
sage:  Q.group_of_mutations().cardinality()
                               322560
```

- `is_finite(self)`
  Returns `True`, iff self is of finite type. This is done by checking if it is mutation-equivalent to a quiver of finite type.

- `is_mutation_finite(self, nr_of_checks=None, return_path=False)`
  Returns `True`, iff self if of finite mutation type. **Warning:** The algorithm is non-deterministic and uses random mutations in various directions. Might theoretically result in a wrong `True` return. The number of checks can be set, the default is 1000 times the number of vertices of self. If `return_path` is `True`, then a path to a non-mutation-finite quiver is returned, if found.

- `mutation_type(self)`
  Returns the mutation type of self if it can be determined.
  - First, it is checked if self is mutation-equivalent to a quiver of a classical type using the descriptions of the classification types,
  - then, it is checked if self is contained in an exceptional mutation class which are hard-coded,
  - if it was not possible to determine the mutation type, it is checked if self is mutation-finite or infinite

  Warning: The algorithm to determine quivers of mutation type $\tilde{D}_n$ (which is `['D',n,1]`) is not yet implemented!

6.4. **ClusterSeed.** The constructor of the class `ClusterSeed` allows the same input as the class `Quiver` to construct a cluster seed. Moreover, many attributes and methods for cluster seeds and for quivers coincide. Often, the cluster seed simply calls the quiver method.

- `QuiverMutationType`
- `list` or `tuple` representing a quiver mutation type
- `ClusterSeed`
- `matrix`: a skew-symmetrizable matrix which represents the exchange matrix
- `Quiver`
- `DiGraph`: the digraph must represent a quiver
- `list` of `tuples` representing the edge list of a digraph for a quiver

The attributes of `ClusterSeed` are given by

- `_M`
  The exchange matrix of self.

- `_cluster`
  The cluster as a list of cluster variables.

- `_n`
  The number of cluster variables (which is the number of columns in the

exchange matrix).

- `_m`
  The number of frozen variables (which is the number of rows − the number of columns in the exchange matrix).

- `_R`
  The base ring in which the cluster variables live.

- `_quiver`
  The quiver attached to self.

- `_description`
  The string representation of self.

- `_mutation_type`
  The mutation type of self, if known, `None` otherwise

The methods of `ClusterSeed` are given by

- `__init__(self, data, frozen=0)`
  Frozen sets the later vertices to be frozen

```
sage: S1 = ClusterSeed([(0,1),(1,2),(2,3)]); S1
        A seed for a cluster algebra of rank 4
sage: Q2 = Quiver([(0,1),(1,2),(2,3)],frozen=1); Q2
 A seed for a cluster algebra of rank 3 with 1 frozen variable
sage: Q1.b_matrix(); Q2.b_matrix()
```

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 \\ 0 & 0 & -1 & 0 \end{pmatrix} \qquad \begin{pmatrix} 0 & 1 & 0 \\ -1 & 0 & 1 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{pmatrix}$$

- `__eq__(self, other)`
  Returns `True`, iff self and other have the same exchange matrix and the same cluster.

- `_repr_(self)`
  Returns the string representation of self

```
sage: S = ClusterSeed(['A',3]); S._repr_()
    "A seed for a cluster algebra of rank 3 of type ['A', 3]"
```

- `plot(self, circular=False, mark=None)`
  Returns a random/circular plot of self with a given marked vertex. Calls the method for quivers.

- `show(self, fig_size=1, circular=False, mark=None)`
  Shows the plot of self.

- `interact(self, fig_size=1, circular=True)`
  Starts an interactive mode, as shown in Figure 1 at the end of Section 3.

- `save_image(self, filename, circular=False)`
  Saves a plot of self to filename.

- `b_matrix(self)`
  Returns the exchange matrix of self.

- `cluster(self)`
  Returns the cluster of self

```
sage: S = ClusterSeed(['A',3]); S.cluster()
```

$$[x_0, x_1, x_2]$$

```
sage: S.mutate(0); S.cluster()
```

$$\left[\frac{x_1+1}{x_0}, x_1, x_2\right]$$

```
sage: S.mutate(1); S.cluster()
```

$$\left[\frac{x_1+1}{x_0}, \frac{x_0x_2+x_1+1}{x_0x_1}, x_2\right]$$

- `ground_field(self)`
  Returns the ground field in which the cluster variables of self live

```
sage: S.ground_field()
```

```
Fraction Field of Multivariate Polynomial Ring
in x0, x1, x2 over Rational Field
```

- `x(self,k)`
  Returns the kth initial cluster variable of self.

- `y(self,k)`
  Returns the kth frozen variable of self.

- `n(self)`
  Returns the number of cluster variables of self.

- `m(self)`
  Returns the number of frozen variables of self.

- `exchangeable_variables(self)`
  Returns a `list` of all cluster variables of self.

- `frozen_variables(self)`
  Returns a `list` of all frozen variables of self.

- `quiver(self)`
  Returns the `Quiver` associated to self.

- `is_acyclic(self)`
  Returns `True`, iff self is acyclic.

- `is_bipartite(self, return_bipartition=False)`
  Returns `True`, iff self is bipartite, if return_bipartition is `True`, the bipartition is returned

```
sage: ClusterSeed(['A',4]).is_bipartite(return_bipartition=True)

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

- `mutate(self, sequence, inplace=True)`
  Mutates at an index or at a list of indices, if inplace is `True`, self is modified, otherwise a new cluster seed is returned.

- `mutation_sequence(self, sequence, show_sequence=False, fig_size=1.2,return_output='seed')`
  Returns a list depending on `return_output` obtained from a sequence of mutations. If show_sequence is `True`, the sequence is shown with a given fig_size. The possible outputs are
  - `'seed'`: a list of cluster seeds is returned
  - `'matrix'`: a list of exchange matrices is returned
  - `'var'`: a list of cluster variables is returned

- `principal_restriction(self)`
  Returns the principal restriction of self. This is obtained from self by deleting all frozen variables.

- `principal_extension(self)`
  Returns the principal extension of self. This can be used only for seeds without frozen variables. Returns a new seed with exchange matrix of size $2n \times n$ given by the exchange matrix of self of size $n \times n$ with an additional identity matrix added below.

- `reorient(self,data)`
  Reorients self by reorienting the corresponding quiver. Calls the method for quivers.

- `set_cluster(self, cluster)`
  Sets the set of clusters of self to `cluster`.

- `reset_cluster(self)`
  Sets the set of clusters of self back to the initial cluster.

- `mutation_class_iter(self, depth=infinity, show_depth=False, return_paths=False, up_to_equivalence=True, only_sink_source=False)`

Returns an iterator which goes through the mutation class of self depending on several parameters
  - `depth`: integer, only quivers with distance at most depth from self are returned
  - `show_depth`: if `True`, the actual depth of the mutation is shown
  - `return_paths`: if `True`, a shortest path of mutation sequences from self to the given quiver is returned as well
  - `up_to_equivalence`: if `True`, only quivers up to equivalence are considered
  - `only_sink_source`: if `True`, only mutations at sinks and sources are applied

- `mutation_class(self, depth=infinity, show_depth=False, return_paths=False, up_to_equivalence=True, only_sink_source=False)`
  Returns a list of all quivers in the corresponding iterator

- `cluster_class_iter(self, depth=infinity, show_depth=False, up_to_equivalence=True)`
  Returns an iterator through all clusters mutation-equivalent to self up to a given depth. Moreover, it is possible to show the actual depth together with several parameters, or to output clusters as labeled seeds.

- `cluster_class(self, depth=infinity, show_depth=False, up_to_equivalence=True)`
  Returns a list of all clusters mutation-equivalent to self up to a given depth. Moreover, it is possible to show the actual depth together with several parameters, or to output cluster as labeled seeds.

- `b_matrix_class_iter(self, depth=infinity, up_to_equivalence=True)`
  Returns an iterator through all matrices mutation-equivalent to self up to a given depth, and up to permutation of rows and columns unless specified otherwise.

- `b_matrix_class(self, depth=infinity, up_to_equivalence=True)`
  Returns a list of all matrices mutation-equivalent to self up to a given depth, and up to permutation of rows and columns unless specified otherwise.

- `variable_class_iter(self, depth=infinity, ignore_bipartite_belt=False)`
  Returns an iterator through all variables obtained from self by mutations up to a given depth. **Warning:** If at some point a bipartite seed is reached, another algorithm is used unless the parameter `ignore_bipartite_belt` is set to be `True`. See the description in Section 4.4.

- `variable_class(self, depth=infinity, ignore_bipartite_belt=False)`

Returns a list of all variables obtained from self by mutations up to a given depth. **Warning:** If at some point a bipartite seed is reached, another algorithm is used unless the parameter `ignore_bipartite_belt` is set to be `True`. See the description in Section 4.4.

- `group_of_mutations(self)`
  Returns the group of mutations of self. **Warning:** The permutation group is very big! This group differs for quivers and for cluster seeds, as different cluster seeds may have the same exchange matrix and thus the same quiver. This group is defined to be the group of permutation given as follows. The ground set is the mutation class of self without taking equivalence of seeds into account, and the group is generated by the n involutions on this set given by mutation at the n different vertices. Observe that the analogous operation on the mutation class up to equivalence does not give a group (this can be easily checked in type A_3). Basically nothing is known about this group.

```
sage: S = ClusterSeed(['A',2])
sage: S.group_of_mutations()

 Permutation Group with generators [(1,2)(3,4)(5,6)(7,9)(8,10),
                  (1,3)(2,5)(4,7)(6,8)(9,10)]

sage: S = ClusterSeed(['B',2])
sage: S.group_of_mutations()

               Permutation Group with generators
               [(1,2)(3,4)(5,6), (1,3)(2,5)(4,6)]

sage: Q = ClusterSeed(['A',1])
sage: Q.group_of_mutations().cardinality()

                                2

sage: Q = ClusterSeed(['A',2])
sage: Q.group_of_mutations().cardinality()

                                10

sage: Q = ClusterSeed(['A',3])
sage: Q.group_of_mutations().cardinality()

                            705438720
```

- `is_finite(self)`
  Returns `True`, iff self is of finite type. Calls the method for the quiver of self.

- `is_mutation_finite(self, nr_of_checks=None, return_path=False)`
  Returns `True`, iff self is of finite mutation type. Calls the method for the quiver of self.

- `mutation_type(self)`
  Returns the mutation type of self, if possible. Calls the method for the quiver of self.

6.5. **ClusterVariable.** By definition, a cluster variable is an element in the field of rational function in n variables[9]. The class `ClusterVariable` provides two extra features for cluster variables:

(1) The connection to almost positive roots in finite types (positive roots are not yet provided in `sage` for affine types).
(2) An ordering for cluster variables which is inspired by its connection to almost positive roots:
   - They are ordered first by total degree of the denominator (in particular, the variables in the initial seed come first in natural order),
   - If the degree is equal and positive, they are ordered lexicographically with $x_0 > x_1 > \ldots > x_{n-1}$.

```
sage: for f in ClusterSeed(['A',2]).variable_class():
....:     print f, f.almost_positive_root()
```

$$\begin{array}{rl} x_0 & -\alpha_1 \\ x_1 & -\alpha_2 \\ (x_1+1)/x_0 & \alpha_1 \\ (x_0+1)/x_1 & \alpha_2 \\ (x_0+x_1+1)/(x_0x_1) & \alpha_1+\alpha_2 \end{array}$$

Two further examples of the ordering can be found in Section 4.2. It is planned to include more functionalities for the cluster variable class in the future.

References

[ASS06] I. Assem, D. Simson, and A Skowronski, Elements of the representation theory of associative algebras. Vol. 1 Techniques of representation theory. London Mathematical Society Student Texts **65**, emphCambridge University Press, Cambridge, 2006.

[Bas10] J. Bastian, Mutation classes of $\tilde{A}_n$-quivers and derived equivalence classification of cluster tilted algebras of type $\tilde{A}_n$, *Algebra & Number Theory* (to appear), 24 pp., eprint, `arXiv:0901.1515`, 2010.

[BPRS10] J. Bastian, T. Prellberg, M. Rubey, and C. Stump, Counting the number of elements in the mutation classes of $\tilde{A}_n$-quivers, eprint, `arXiv:0906.0487`, 2010.

[BFZ05] A. Berenstein, S. Fomin, and A. Zelevinsky, Cluster algebras III: Upper bounds and Double-Bruhat cells , *Duke Mathematical Journal* **126**, no. 1, 1–52, 2005.

[B-MPW09] M. Bosquet-Melou, J. Propp, and J. West, Perfect matchings for the three-term Gale-Robinson sequences, eprint, `arXiv:0906.3125`, 2009.

[BT94] R. Bott and C. Taubes, On the self-linking of knots. Topology and physics, *J. Math. Phys.* **35**, no. 10, 5247-5287, 1994.

[BT09] A. B. Buan and H. A. Torkildsen, The number of elements in the mutation class of a quiver of type D_n, *Electron. J. Combin.* **16**, no. 1, 2009.

[CC06] P. Caldero and F. Chapoton, Cluster algebras as Hall algebras of quiver representations, *Comment. Math. Helv.* **81** , 595-616, 2006.

[CCS06] P. Caldero, F. Chapoton, R. Schiffler, Quivers with relations arising from clusters (A_n case), *Trans. Amer. Math. Soc.* **358** , no. 3, 1347-1364, 2006.

[CK08] P. Caldero and B. Keller, From triangulated categories to cluster algebras. *Invent. Math.* **172**, no. 1, 169-211, 2008.

[CR08] P. Caldero and M. Reineke, On the quiver Grassmannian in the acyclic case, *Journ. Pure Appl. Alg.* **212**, no. 11, 2369–2380, 2008.

[9] Moreover, by Theorem 1.1, they are actually multivariate Laurent polynomials in n variables, although for the moment we do not use this functionality.

[CZ06] P. Caldero and A. Zelevinsky, Laurent expansions in cluster algebras via quiver representations, *Mosc. Math. J.* **6**, no. 3, 411–429, 2006.

[CP03] G. Carroll and G. Price, Two new combinatorial models for the Ptolemy recurrence, unpublished memo, 2003.

[CS04] G. Carroll and D. Speyer, The cube recurrence. *Electron. J. Combin.* **11**, no. 1, Research Paper 73, 31 pp, 2004.

[Cha] F. Chapoton, Mutations in Maple and Mupad, `http://math.univ-lyon1.fr/~chapoton/mode_emploi.html`

[CFZ02] F. Chapoton, S. Fomin and A. Zelevinsky, Polytopal realizations of generalized associahedra, *Canadian Mathematical Bulletin* **45**, no. 4 537-566, 2002.

[Dem09] L. Demonet, Categorification of skew-symmetrizable cluster algebras, eprint, `arXiv:0909.1633`, 2009.

[DWZ09] H. Derksen, J. Weyman and A. Zelevinsky, Quivers with potentials and their representations II: Applications to cluster algebras, eprint, `arXiv:09040676`, 2009.

[DO08] H. Derksen and T. Owen, New Graphs of Finite Mutation Type, *Electron. J. Combin.* **15**, no. 1, Research Paper 139, 15 pp, 2008.

[DiFK08] P. Di Francesco and R. Kedem, Q-systems, Heaps, Paths and Cluster Positivity, eprint `arXiv:0811.3027`, 2008.

[DiFK09a] P. Di Francesco and R. Kedem, Q-system Cluster Algebras, Paths and Total Positivity, eprint, `arXiv:0906.3421`, 2009.

[DiFK09b] P. Di Francesco and R. Kedem, Positivity of the T-system cluster algebra, eprint, `arXiv:0908.3122`, 2009.

[DiFK09c] P. Di Francesco and R. Kedem, Discrete non-commutative integrability: the proof of a conjecture by M. Kontsevich, eprint, `arXiv:0909.0615`, 2009.

[DR76] V. Dlab and M. Ringel, Indecomposable representations of graphs and algebras, *Mem. Amer. Math. Soc.* **6**, no. 173, 1976.

[Dup08] G. Dupont, An approach to non-simply laced cluster algebras, *J. Algebra* **320**, no. 4, 1626–1661, 2008.

[Dup09] G. Dupont, Positivity in coefficient-free rank two cluster algebras, *Electron. J. Combin.* **16**, no. 1, Research Paper 98, 11 pp, 2009.

[DP] G. Dupont and M. Pérotin, Quiver Mutation Explorer, `http://pages.usherbrooke.ca/gdupont2/QME/blog/who-is-qme/`

[DP10] G. Dupont and M. Pérotin, Private communications, 2010.

[EKLP92] N. Elkies, G. Kuperberg, M. Larsen, J. Propp, Alternating-sign matrices and domino tilings I, *J. Algebraic Combin.* **1**, no. 2, 111-132, 1992.

[FST08] A. Felikson, M. Shapiro, P. Tumarkin. Skew-symmetric cluster algebras of finite mutation type, eprint, `arXiv:0811.1703`, 2008.

[FST10] A. Felikson, M. Shapiro, P. Tumarkin. Cluster algebras of finite mutation type via unfoldings, eprint, `arxiv:1006.4276`, 2010.

[FG06] V. Fock and A. Goncharov, Moduli spaces of local systems and higher Teichmüller theory. *Publ. Math. Inst. Hautes Études Sci.* No. 103, 1–211, 2006.

[FG07] V. Fock and A. Goncharov, Dual Teichmüller and lamination spaces. Handbook of Teichmüller theory. Vol. I, 647–684, IRMA Lect. Math. Theor. Phys., 11, Eur. Math. Soc., Zürich, 2007.

[FG09] V. Fock and A. Goncharov, Cluster ensembles, quantization and the dilogarithm, *Ann. Sci. Ecole Normale. Sup.*, (4) **42**, no. 6 865-930, 2009.

[FST08] S. Fomin, M. Shapiro, and D. Thurston, Cluster algebras and triangulated surfaces. Part I: Cluster complexes, *Acta Math.* **201**, 83-146, 2008.

[FT08] S. Fomin and D. Thurston, Cluster algebras and triangulated surfaces. Part II: Lambda Lengths, eprint, `http://www.math.lsa.umich.edu/~fomin/Papers/cats2.ps`, 2008.

[FZ99] S. Fomin and A. Zelevinsky, Double Bruhat Cells and Total Positivity, *Journal of the American Mathematical Society* **12**, no 2, 335-380, 1999.

[FZ00] S. Fomin and A. Zelevinsky, Total Positivity: tests and parametrizations, *Math. Intelligencer* **22** , no 1, 22-33, 2000.

[FZ02a] S. Fomin and A. Zelevinsky, Cluster algebras I: Foundations, *J. Amer. Math. Soc.* **15** , 497-529, 2002.

[FZ02b] S. Fomin and A. Zelevinsky, The Laurent Phenomenon, *Adv. in Applied Math.* **28** , 119-144, 2002.

[FZ03a] S. Fomin and A. Zelevinsky, Y-systems and generalized associahedra, *Ann. of Math.* **158** , 977-1018, 2003.

[FZ03b] S. Fomin and A. Zelevinsky, Cluster algebras II: Finite type classification, *Invent. Math.* **154** , 63-121, 2003.

[FZ07] S. Fomin and A. Zelevinsky, Cluster algebras IV: Coefficients, *Compositio Mathematica* **143** , 112-164, 2007.

[FM09] A. Fordy and R. Marsh, Cluster mutation-periodic quivers and associated Laurent sequences, eprint, `arXiv:0904.0200`, 2009.

[Gal91] D. Gale, The strange and surprising saga of the Somos sequences, *Math. Intelligencer* **13** , no. 1, 40-43, 1991.

[GSV05] M. Gekhtman, M. Shapiro and A. Vainshtein, Cluster algebras and Weil-Petersson forms, Duke Math. J. **127**, 291–311, 2005.

[GSV10] M. Gekhtman, M. Shaprio and A. Vainshtein, Cluster algebras and Poisson Geometry, *AMS Mathematical Surveys and Monographs* **167**, 2010.

[Hen09] T. Henrich, Mutation-classes of diagrams via infinite graphs, to appear in *Math. Nachr.*, `arXiv:0903.1924`, 2009.

[Hum72] J. E. Humphreys, Introduction to Lie algebras and representation theory, *Graduate Texts in Mathematics* **9**, Springer-Verlag, 1972.

[Kac94] V. Kac Infinite Dimensional Lie Algebras, *Cambridge University Press*, 1994.

[McK81] B.D. McKay, Practical Graph Isomorphism, *Congressus Numerantium* **30**, 45–87, 1981.

[Kel] B. Keller, Quiver mutation in Java, `http://people.math.jussieu.fr/∼keller/quivermutation`

[Kel2] B. Keller, Cluster algebras, quiver representations, and triangulated categories. *Triangulated categories*, 76-160, *London Math. Soc. Lecture Note Ser.,* 375, Cambridge Univ. Press, Cambridge, 2010.

[Lus93] G. Lusztig, *Introduction to quantum groups*, Birkhäuser, Boston, 1993.

[Mus02] G. Musiker, *Cluster algebras, Somos sequences, and exchange graphs*, Senior Thesis, Harvard, `http://math.mit.edu/∼ musiker/uthesis.pdf`, 2002.

[Mus08] G. Musiker, A graph theoretic expansion formula for cluster algebras of classical type, to appear in *Ann. of Combin.*, `arXiv:0710.3574`, 2008.

[MP07] G. Musiker and J. Propp, Combinatorial interpretations for rank-two cluster algebras of affine type. *Electron. J. Combin.* **14**, no. 1, Research Paper 15, 23pp, 2007.

[MS08] G. Musiker and R. Schiffler, Cluster expansion formulas and perfect matchings, to appear in *J. Algebraic Combin.*, `arXiv:0810.3638`, 2008.

[MSW09] G. Musiker, R. Schiffler, and L. Williams. Positivity of cluster algebras from surfaces, eprint, `arXiv:0906.0748`, 2009.

[Nak09] H. Nakajima, Quiver varieties and cluster algebras, eprint, `arXiv:0905.0002`, 2009.

[Qin] F. Qin, Quantum Cluster Variables via Serre Polynomials, eprint, `arXiv:1004.4171`, 2010

[Pro93] J. Propp, Lattice structure for orientations of graphs, eprint, `arXiv:math/0209.5005`, 1993.

[Pro08] J. Propp, The combinatorics of frieze patterns and Markoff numbers, eprint, `arXiv:math.CO/0511633`, 2008.

[Sch08] R. Schiffler, On cluster algebras arising from unpunctured surfaces II, eprint, `arXiv:0809.2593`, 2008.

[ST09] R. Schiffler and H. Thomas : On cluster algebras arising from unpunctured surfaces, *Int. Math. Res. Notices.*, 2009.

[Sco06] J. Scott, Grassmanians and Cluster Algebras, *Proc. London Math. Society* **92**, 345-280, 2006

[SZ04] P. Sherman and A. Zelevinsky, Positivity and canonical bases in rank 2 cluster algebras of finite and affine types. Mosc. Math. J. **4**, no. 4, 947–974, 982, 2004.

[Spe07] D. Speyer, Perfect Matchings and the Octahedron Recurrence, *J. Algebraic Combin.*, **25**, no. 3, 309–348, 2007.

[Sage] W.A. Stein and others, Sage Mathematics Software, *The Sage Development Team*, `http://www.sagemath.org`, 2011.

[SageComb] The Sage-Combinat community, Sage-Combinat: enhancing Sage as a toolbox for computer exploration in algebraic combinatorics, `http://combinat.sagemath.org`, 2011.

[Stu11] C. Stump, Mutation classes of non-simply-laced quivers. (in preparation)

[Tor08] H.A. Torkildsen, Counting cluster-tilted algebras of type A_n,*International Electronic Journal of Algebra* **4**, 149–158, 2008.

[Vat08] D. Vatne, The mutation class of Dn quivers, *Comm. Algebra* (to appear), eprint, arXiv: 0810.4789, 2008.

[Zel02] A. Zelevinsky, From Littlewood coefficients to cluster algebras in three lectures, Symmetric Functions 2001: Surveys of Developments and Perspectives, Proceedings of the NATO Advanced Study Institute 2001, S.Fomin (Ed.), 253-273; NATO Science Series II: Mathematics, Physics and Chemistry, Vol. 74. Kluwer Academic Publishers, Dordrecht, 2002.

[Zel07] A. Zelevinsky, Semicanonical basis generators of the cluster algebra of type $A_1^{(1)}$. *Electron. J. Combin.* **14**, no. 1, Note 4, 5 pp, 2007.

School of Mathematics, University of Minnesota, Minneapolis, MN 55455, USA
E-mail address: musiker@math.umn.edu
URL: http://www.math.umn.edu/∼musiker

LaCIM, Université du Québec à Montréal, Montréal (Québec), Canada
E-mail address: christian.stump@lacim.ca
URL: http://homepage.univie.ac.at/christian.stump/