Thematic Tutorials

*Release 9.7*

The Sage Development Team

Sep 20, 2022
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1  Introduction to Sage</td>
<td>3</td>
</tr>
<tr>
<td>2  Introduction to Python</td>
<td>5</td>
</tr>
<tr>
<td>3  Calculus and Plotting</td>
<td>7</td>
</tr>
<tr>
<td>4  Algebra</td>
<td>9</td>
</tr>
<tr>
<td>5  Number Theory</td>
<td>11</td>
</tr>
<tr>
<td>6  Geometry</td>
<td>13</td>
</tr>
<tr>
<td>7  Combinatorics</td>
<td>15</td>
</tr>
<tr>
<td>8  Algebraic Combinatorics</td>
<td>17</td>
</tr>
<tr>
<td>9  Parents/Elements, Categories and Algebraic Structures</td>
<td>19</td>
</tr>
<tr>
<td>10 Numerical Computations</td>
<td>21</td>
</tr>
<tr>
<td>11 Advanced Programming</td>
<td>23</td>
</tr>
<tr>
<td>12 Documentation</td>
<td>25</td>
</tr>
<tr>
<td>Bibliography</td>
<td>517</td>
</tr>
</tbody>
</table>
Here you will find Sage demonstrations, quick reference cards, primers, and thematic tutorials,

- A quickref (or quick reference card) is a one page document with the essential examples, and pointers to the main entry points.
- A primer is a document meant for a user to get started by himself on a theme in a matter of minutes.
- A tutorial is more in-depth and could take as much as an hour or more to get through.

This documentation is licensed under a Creative Commons Attribution-Share Alike 3.0 License.
CHAPTER
ONE

INTRODUCTION TO SAGE

• Introductory Sage Tutorial (PREP)
• Sage’s main tutorial
Chapter 1. Introduction to Sage
CHAPTER TWO

INTRODUCTION TO PYTHON

• Tutorial: Sage Introductory Programming (PREP)
• Tutorial: Programming in Python and Sage
• Tutorial: Comprehensions, Iterators, and Iterables
• Tutorial: Objects and Classes in Python and Sage
• Functional Programming for Mathematicians
CHAPTER
THREE

CALCULUS AND PLOTTING

• Tutorial: Symbolics and Plotting (PREP)
• Tutorial: Calculus (PREP)
• Tutorial: Advanced-2D Plotting (PREP)
• Tutorial: Vector Calculus in Euclidean Spaces
CHAPTER
FOUR

ALGEBRA

• Group Theory and Sage
• Lie Methods and Related Combinatorics in Sage
• Tutorial: Using free modules and vector spaces
• Steenrod Algebra Modules
CHAPTER
FIVE

NUMBER THEORY

• *Number Theory and the RSA Public Key Cryptosystem*
• Introduction to the p-adics
• *Three Lectures about Explicit Methods in Number Theory Using Sage*
• Polyhedra
CHAPTER
SEVEN

COMBINATORICS

• Introduction to combinatorics in Sage
• Coding Theory in Sage
• How to write your own classes for coding theory
ALGEBRAIC COMBINATORICS

• Algebraic Combinatorics in Sage
• Tutorial: Symmetric Functions
• Lie Methods and Related Combinatorics in Sage
• Tutorial: visualizing root systems
• Abelian Sandpile Model
PARENTS/ELEMENTS, CATEGORIES AND ALGEBRAIC STRUCTURES

• *How to implement new algebraic structures in Sage*
• Elements, parents, and categories in Sage: a primer
• Implementing a new parent: a tutorial
• *Tutorial: Implementing Algebraic Structures*
• Numerical Computing with Sage
• Linear Programming (Mixed Integer)
• How to call a C code (or a compiled library) from Sage?
• Profiling in Sage
12.1 Thematic tutorial document tree

12.1.1 Algebraic Combinatorics in Sage

Author: Anne Schilling (UC Davis)

These notes provide some Sage examples for Stanley’s book:

A free pdf version of the book without exercises can be found on Stanley’s homepage.

Preparation of this document was supported in part by NSF grants DMS–1001256 and OCI–1147247.

I would like to thank Federico Castillo (who wrote a first version of the $n$-cube section) and Nicolas M. Thiery (who wrote a slightly different French version of the Tsetlin library section) for their help.

Walks in graphs

This section provides some examples on Chapter 1 of Stanley’s book [Stanley2013].

We begin by creating a graph with 4 vertices:

```sage
g = Graph(4)
g
```

Graph on 4 vertices

This graph has no edges yet:

```sage
g.vertices(sort=True)
[0, 1, 2, 3]
g.edges(sort=True)
[]
```

Before we can add edges, we need to tell Sage that our graph can have loops and multiple edges.:

```sage
g.allow_loops(True)
g.allow_multiple_edges(True)
```

Now we are ready to add our edges by specifying a tuple of vertices that are connected by an edge. If there are multiple edges, we need to add the tuple with multiplicity.:
Thematic Tutorials, Release 9.7

```
sage: G.add_edges([(0,0),(0,0),(0,1),(0,3),(1,3),(1,3)])
```

Now let us look at the graph!

```
sage: G.plot()
```

Graphics object consisting of 11 graphics primitives

We can construct the adjacency matrix:

```
sage: A = G.adjacency_matrix()
sage: A
```

\[
\begin{bmatrix}
2 & 1 & 0 & 1 \\
1 & 0 & 0 & 2 \\
0 & 0 & 0 & 0 \\
1 & 2 & 0 & 0
\end{bmatrix}
\]

The entry in row \(i\) and column \(j\) of the \(\ell\)-th power of \(A\) gives us the number of paths of length \(\ell\) from vertex \(i\) to vertex \(j\). Let us verify this:

```
sage: A**2
```

\[
\begin{bmatrix}
6 & 4 & 0 & 4 \\
4 & 5 & 0 & 1 \\
0 & 0 & 0 & 0 \\
4 & 1 & 0 & 5
\end{bmatrix}
\]

There are 4 paths of length 2 from vertex 0 to vertex 1: take either loop at 0 and then the edge (0, 1) (2 choices) or take the edge (0, 3) and then either of the two edges (3, 1) (two choices):

```
sage: (A**2)[0,1]
```

4

To count the number of closed walks, we can also look at the sum of the \(\ell\)-th powers of the eigenvalues. Even though the eigenvalues are not integers, we find that the sum of their squares is an integer:

```
sage: A.eigenvalues()
sage: sum(la**2 for la in A.eigenvalues())
```

16.00000000000000

We can achieve the same by looking at the trace of the \(\ell\)-th power of the matrix:
This section provides some examples on Chapter 2 of Stanley’s book [Stanley2013], which deals with \( n \)-cubes, the Radon transform, and combinatorial formulas for walks on the \( n \)-cube.

The vertices of the \( n \)-cube can be described by vectors in \( \mathbb{Z}^n_2 \). First we define the addition of two vectors \( u, v \in \mathbb{Z}^n_2 \) via the following distance:

\[
\text{sage: def dist(u,v):
    h = [(u[i]+v[i])\%2 for i in range(len(u))]
    return sum(h)
}
\]

The distance function measures in how many slots two vectors in \( \mathbb{Z}^n_2 \) differ:

\[
\text{sage: u = (1,0,1,1,1,0)
\text{sage: v = (0,0,1,1,0,0)
\text{sage: dist(u,v)
2}
}\]

Now we are going to define the \( n \)-cube as the graph with vertices in \( \mathbb{Z}^n_2 \) and edges between vertex \( u \) and vertex \( v \) if they differ in one slot, that is, the distance function is 1:

\[
\text{sage: def cube(n):
    G = Graph(2**n)
    vertices = Tuples([0,1],n)
    for i in range(2**n):
        for j in range(2**n):
            if dist(vertices[i],vertices[j]) == 1:
                G.add_edge(i,j)
    return G
}\]

We can plot the 3 and 4-cube:

\[
\text{sage: cube(3).plot()}
\]

Graphics object consisting of 21 graphics primitives
Next we can experiment and check Corollary 2.4 in Stanley’s book, which states the $n$-cube has $n$ choose $i$ eigenvalues equal to $n - 2i$:

```
sage: G = cube(2)
sage: G.adjacency_matrix().eigenvalues()
[2, -2, 0, 0]
sage: G = cube(3)
sage: G.adjacency_matrix().eigenvalues()
[3, -3, 1, 1, -1, -1, -1]
sage: G = cube(4)
sage: G.adjacency_matrix().eigenvalues()
[4, -4, 2, 2, 2, 2, -2, -2, -2, -2, 0, 0, 0, 0, 0, 0]
```

It is now easy to slightly vary this problem and change the edge set by connecting vertices $u$ and $v$ if their distance is 2 (see Problem 4 in Chapter 2):

```
sage: def cube_2(n):
    ....:    G = Graph(2**n)
    ....:    vertices = Tuples([0,1],n)
    ....:    for i in range(2**n):
    ....:        for j in range(2**n):
    ....:            if dist(vertices[i],vertices[j]) == 2:
    ....:                G.add_edge(i,j)
    ....:    return G

sage: G = cube_2(2)
sage: G.adjacency_matrix().eigenvalues()
[1, 1, -1, -1]
sage: G = cube_2(4)
sage: G.adjacency_matrix().eigenvalues()
[6, 6, -2, -2, -2, -2, -2, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```
Note that the graph is in fact disconnected. Do you understand why?

```
sage: cube_2(4).plot()
Graphics object consisting of 65 graphics primitives
```

The Tsetlin library

Introduction

In this section, we study a simple random walk (or Markov chain), called the Tsetlin library. It will give us the opportunity to see the interplay between combinatorics, linear algebra, representation theory and computer exploration, without requiring heavy theoretical background. I hope this encourages everyone to play around with this or similar systems and investigate their properties! Formal theorems and proofs can be found in the references at the end of this section.

It has been known for several years that the theory of group representations can facilitate the study of systems whose evolution is random (Markov chains), breaking them down into simpler systems. More recently it was realized that generalizing this (namely replacing the invertibility axiom for groups by other axioms) explains the behavior of other particularly simple Markov chains such as the Tsetlin library.

The Tsetlin library

Consider a bookshelf in a library containing $n$ distinct books. When a person borrows a book and then returns it, it gets placed back on the shelf to the right of all books. This is what we naturally do with our pile of shirts in the closet: after use and cleaning, the shirt is placed on the top of its pile. Hence the most popular books/shirts will more likely appear on the right/top of the shelf/pile.

This type of organization has the advantage of being self-adaptive:

- The books most often used accumulate on the right and thus can easily be found.
- If the use changes over time, the system adapts.

In fact, this type of strategy is used not only in everyday life, but also in computer science. The natural questions that arise are:

- *Stationary distribution*: To which state(s) does the system converge to? This, among other things, is used to evaluate the average access time to a book.
- *The rate of convergence*: How fast does the system adapt to a changing environment?

Let us formalize the description. The Tsetlin library is a discrete Markov chain (discrete time, discrete state space) described by:

- The state space $\Omega_n$ is given by the set of all permutations of the $n$ books.
The transition operators are denoted by $\partial_i : \Omega_n \rightarrow \Omega_n$. When $\partial_i$ is applied to a permutation $\sigma$, the number $i$ is moved to the end of the permutation.

We assign parameters $x_i \geq 0$ for all $1 \leq i \leq n$ with $\sum_{i=1}^{n} x_i = 1$. The parameter $x_i$ indicates the probability of choosing the operator $\partial_i$.

**Transition graph and matrix**

One can depict the action of the operators $\partial_i$ on the state space $\Omega_n$ by a digraph. The following picture shows the action of $\partial_1, \partial_2, \partial_3$ on $\Omega_3$:

![Transition graph](image)

The above picture can be reproduced in Sage as follows:

```sage
P = Poset(((1,2,3),[]))
This is the antichain poset. Its linear extensions are all permutations of \{1, 2, 3\}:

```sage
L = P.linear_extensions()
sage: L
The set of all linear extensions of Finite poset containing 3 elements
```sage
L.list()
[[[3, 2, 1], [3, 1, 2], [1, 3, 2], [1, 2, 3], [2, 1, 3], [2, 3, 1]]

The graph is produced via:

```sage
G = L.markov_chain_digraph(labeling='source'); G
Looped multi-digraph on 6 vertices
```sage
view(G) # not tested

We can now look at the transition matrix and see whether we notice anything about its eigenvalue and eigenvectors:

```sage
M = L.markov_chain_transition_matrix(labeling='source')
sage: M
[[-x0 - x1 x2 0 0 x2 0]
 [ x1 -x0 - x2 x1 0 0 0]
 [ 0 0 -x0 - x1 x2 0 x2]
 [ x0 0 x0 -x1 - x2 0 0]
 [ 0 0 0 x1 -x0 - x2 x1]
 [ 0 x0 0 0 x0 -x1 - x2]
```
This matrix is normalized so that all columns add to 0. So we need to add \((x_0 + x_1 + x_2)\) times the \(6 \times 6\) identity matrix to get the probability matrix:

\[
\begin{bmatrix}
x_2 & x_2 & 0 & 0 & x_2 & 0 \\
x_1 & x_1 & x_1 & 0 & 0 & 0 \\
0 & 0 & x_2 & x_2 & 0 & x_2 \\
x_0 & 0 & x_0 & x_0 & 0 & 0 \\
0 & 0 & 0 & x_1 & x_1 & x_1 \\
0 & x_0 & 0 & 0 & x_0 & x_0 \\
\end{bmatrix}
\]

Since the \(x_i\) are formal variables, we need to compute the eigenvalues and eigenvectors in the symbolic ring \(SR\):

\[
\text{sage: Mt.change_ring(SR).eigenvalues()}
\]

\[
[x_2, x_1, x_0, x_0 + x_1 + x_2, 0, 0]
\]

Do you see any pattern? In fact, if you start playing with bigger values of \(n\) (the size of the underlying permutations), you might observe that there is an eigenvalue for every subset \(S\) of \(\{1, 2, \ldots, n\}\) and the multiplicity is given by a derangement number \(d_{n-|S|}\). Derangement numbers count permutations without fixed point. For the eigenvectors we obtain:

\[
\text{sage: Mt.change_ring(SR).eigenvectors_right()}
\]

\[
[(x_2, [(1, 0, -1, 0, 0, 0)], 1),
(x_1, [(0, 1, 0, 0, -1, 0)], 1),
(x_0, [(0, 0, 1, 1, 0, -1)], 1),
(x_0 + x_1 + x_2, 
(1, 
(x_0 + x_1)/(x_0 + x_2),
 x_0/x_1, 
(x_0^2 + x_0*x_1)/(x_1^2 + x_1*x_2),
(x_0^2 + x_0*x_1)/(x_0^2 + 2*x_2),
(x_0^2 + x_0*x_1)/(x_1^2 + x_2^2)], 1),
(0, [(1, 0, -1, 0, -1, 1), (0, 1, -1, 1, -1, 0)], 2)]
\]

The stationary distribution is the eigenvector of eigenvalues \(1 = x_0 + x_1 + x_2\). Do you see a pattern?

Optional exercises: Study of the transition operators and graph

Instead of using the methods that are already in Sage, try to build the state space \(\Omega_n\) and the transition operators \(\partial_i\) yourself as follows.

1. For technical reasons, it is most practical in Sage to label the \(n\) books in the library by \(0, 1, \ldots, n-1\), and to represent each state in the Markov chain by a permutation of the set \(\{0, \ldots, n-1\}\) as a tuple. Construct the state space \(\Omega_n\) as:

\[
\text{sage: list(map(tuple, Permutations(range(3))))}
\]

\[
[(0, 1, 2), (0, 2, 1), (1, 0, 2), (1, 2, 0), (2, 0, 1), (2, 1, 0)]
\]

2. Write a function \(\text{transition_operator}(\text{sigma}, \text{i})\) which implements the operator \(\partial_i\) which takes as input a tuple \(\text{sigma}\) and integer \(i \in \{1, 2, \ldots, n\}\) and outputs a new tuple. It might be useful to extract subtuples \((\text{sigma}[i:j])\) and concatenation.
3. Write a function `tsetlin_digraph(n)` which constructs the (multi digraph) as described as shown above. This can be achieved using `DiGraph`.

4. Verify for which values of $n$ the digraph is strongly connected (i.e., you can go from any vertex to any other vertex by going in the direction of the arrow). This indicates whether the Markov chain is irreducible.

**Conclusion**

The Tsetlin library was studied from the viewpoint of monoids in [Bidigare1997] and [Brown2000]. Precise statements of the eigenvalues and the stationary distribution of the probability matrix as well as proofs of the statements are given in these papers. Generalizations of the Tsetlin library from the antichain to arbitrary posets was given in [AKS2013].

**Young’s lattice and the RSK algorithm**

This section provides some examples on Young’s lattice and the RSK (Robinson-Schensted-Knuth) algorithm explained in Chapter 8 of Stanley’s book [Stanley2013].

**Young’s Lattice**

We begin by creating the first few levels of Young’s lattice $\mathcal{Y}$. For this, we need to define the elements and the order relation for the poset, which is containment of partitions:

```sage
sage: level = 6
sage: elements = [b for n in range(level) for b in Partitions(n)]
sage: ord = lambda x,y: y.contains(x)
sage: Y = Poset((elements,ord), facade=True)
sage: H = Y.hasse_diagram()
```

The resulting image looks best when `dot2tex` is installed:

```sage
sage: view(H)  # not tested
```
We can now define the up and down operators $U$ and $D$ on $QY$. First we do so on partitions, which form a basis for $QY$:

```
sage: QQY = CombinatorialFreeModule(QQ,elements)

sage: def U_on_basis(la):
....:    covers = Y.upper_covers(la)
....:    return QQY.sum_of_monomials(covers)

sage: def D_on_basis(la):
....:    covers = Y.lower_covers(la)
....:    return QQY.sum_of_monomials(covers)
```

As a shorthand, one also can write the above as:

```
sage: U_on_basis = QQY.sum_of_monomials * Y.upper_covers
sage: D_on_basis = QQY.sum_of_monomials * Y.lower_covers
```

Here is the result when we apply the operators to the partition $(2,1)$:

```
sage: la = Partition([2,1])
sage: U_on_basis(la)
B[[2, 1, 1]] + B[[2, 2]] + B[[3, 1]]
sage: D_on_basis(la)
B[[1, 1]] + B[[2]]
```

Now we define the up and down operator on $QY$:

```
sage: U = QQY.module_morphism(U_on_basis)
sage: D = QQY.module_morphism(D_on_basis)
```

We can check the identity $D_{i+1}U_i - U_{i-1}D_i = I_i$ explicitly on all partitions of $i = 3$:
We can also check that the coefficient of $\lambda \vdash n$ in $U_n(\emptyset)$ is equal to the number of standard Young tableaux of shape $\lambda$:

```
sage: u = QQY(Partition(()))
sage: for i in range(4):
    ....:     u = U(u)
sage: u
B[[1, 1, 1, 1]] + 3*B[[2, 1, 1]] + 2*B[[2, 2]] + 3*B[[3, 1]] + B[[4]]
```

For example, the number of standard Young tableaux of shape $(2, 1, 1)$ is 3:

```
sage: StandardTableaux([2,1,1]).cardinality()
3
```

We can test this in general:

```
sage: for la in u.support():
    ....:     assert u[la] == StandardTableaux(la).cardinality()
```

We can also check this against the hook length formula (Theorem 8.1):

```
sage: def hook_length_formula(p):
    ....:     n = p.size()
    ....:     return factorial(n) // prod(p.hook_length(*c) for c in p.cells())
sage: for la in u.support():
    ....:     assert u[la] == hook_length_formula(la)
```

**RSK Algorithm**

Let us now turn to the RSK algorithm. We can verify Example 8.12 as follows:

```
sage: p = Permutation([4,2,7,3,6,1,5])
sage: RSK(p)
[[[1, 3, 5], [2, 6], [4, 7]], [[1, 3, 5], [2, 4], [6, 7]]]
```

The tableaux can also be displayed as tableaux:

```
sage: P,Q = RSK(p)
sage: P.pp()
1 3 5
2 6
4 7
sage: Q.pp()
1 3 5
2 4
6 7
```

The inverse RSK algorithm is implemented as follows:
We can verify that the RSK algorithm is a bijection:

```python
sage: def check_RSK(n):
    ....:     for p in Permutations(n):
    ....:         assert RSK_inverse(*RSK(p), output='permutation') == p
sage: for n in range(5):
    ....:     check_RSK(n)
```

12.1.2 Abelian Sandpile Model

Author: David Perkinson, Reed College

Introduction

These notes provide an introduction to Dhar’s abelian sandpile model (ASM) and to Sage Sandpiles, a collection of tools in Sage for doing sandpile calculations. For a more thorough introduction to the theory of the ASM, the papers Chip-Firing and Rotor-Routing on Directed Graphs [H] by Holroyd et al., and Riemann-Roch and Abel-Jacobi Theory on a Finite Graph by Baker and Norine [BN] are recommended. See also [PPW2013].

To describe the ASM, we start with a sandpile graph: a directed multigraph $\Gamma$ with a vertex $s$ that is accessible from every vertex. By multigraph, we mean that each edge of $\Gamma$ is assigned a nonnegative integer weight. To say $s$ is accessible from some vertex $v$ means that there is a sequence (possibly empty) of directed edges starting at $v$ and ending at $s$. We call $s$ the sink of the sandpile graph, even though it might have outgoing edges, for reasons that will be made clear in a moment.

We denote the vertex set of $\Gamma$ by $V$, and define $\tilde{V} = V \setminus \{s\}$. For any vertex $v$, we let out-degree($v$) (the out-degree of $v$) be the sum of the weights of all edges leaving $v$.

Configurations and divisors

A configuration on $\Gamma$ is an element of $\mathbb{Z}_{\geq 0} \tilde{V}$, i.e., the assignment of a nonnegative integer to each nonsink vertex. We think of each integer as a number of grains of sand being placed at the corresponding vertex. A divisor on $\Gamma$ is an element of $\mathbb{Z}V$, i.e., an element in the free abelian group on all of the vertices. In the context of divisors, it is sometimes useful to think of a divisor on $\Gamma$ as assigning dollars to each vertex, with negative integers signifying a debt.

Stabilization

A configuration $c$ is stable at a vertex $v \in \tilde{V}$ if $c(v) < \text{out-degree}(v)$. Otherwise, $c$ is unstable at $v$. A configuration $c$ (in itself) is stable if it is stable at each nonsink vertex. Otherwise, $c$ is unstable. If $c$ is unstable at $v$, the vertex $v$ can be fired (toppled) by removing out-degree($v$) grains of sand from $v$ and adding grains of sand to the neighbors of $v$, determined by the weights of the edges leaving $v$ (each vertex $w$ gets as many grains as the weight of the edge $(v, w)$ is, if there is such an edge). Note that grains that are added to the sink $s$ are not counted (i.e., they “disappear”).

Despite our best intentions, we sometimes consider firing a stable vertex, resulting in a “configuration” with a negative amount of sand at that vertex. We may also reverse-fire a vertex, absorbing sand from the vertex’s neighbors (including the sink).

Example. Consider the graph:
All edges have weight 1 except for the edge from vertex 1 to vertex 3, which has weight 2. If we let \( c = (5, 0, 1) \) with the indicated number of grains of sand on vertices 1, 2, and 3, respectively, then only vertex 1, whose out-degree is 4, is unstable. Firing vertex 1 gives a new configuration \( c' = (1, 1, 3) \). Here, 4 grains have left vertex 1. One of these has gone to the sink vertex (and forgotten), one has gone to vertex 2, and two have gone to vertex 3, since the edge from 1 to 3 has weight 2. Vertex 3 in the new configuration is now unstable. The Sage code for this example follows.

```sage
sage: g = {0:{}, ...
....: 1:{0:1, 2:1, 3:2}, ...
....: 2:{1:1, 3:1}, ...
....: 3:{1:1, 2:1}}
sage: S = Sandpile(g, 0)  # create the sandpile
sage: S.show(edge_labels=true)  # display the graph
```

Create the configuration:

```sage
sage: c = SandpileConfig(S, {1:5, 2:0, 3:1})
sage: S.out_degree()
{0: 0, 1: 4, 2: 2, 3: 2}
```

Fire vertex 1:

```sage
sage: c.fire_vertex(1)
{1: 1, 2: 1, 3: 3}
```

The configuration is unchanged:
Repeatedly fire vertices until the configuration becomes stable:

```
sage: c.stabilize()
{1: 2, 2: 1, 3: 1}
```

Alternatives:

```
sage: ~c  # shorthand for c.stabilize()
{1: 2, 2: 1, 3: 1}
sage: c.stabilize(with_firing_vector=true)
[{{1: 2, 2: 1, 3: 1}, {1: 2, 2: 2, 3: 3}}]
```

Since vertex 3 has become unstable after firing vertex 1, it can be fired, which causes vertex 2 to become unstable, etc. Repeated firings eventually lead to a stable configuration. The last line of the Sage code, above, is a list, the first element of which is the resulting stable configuration, \( (2, 1, 1) \). The second component records how many times each vertex fired in the stabilization.

Since the sink is accessible from each nonsink vertex and never fires, every configuration will stabilize after a finite number of vertex-firings. It is not obvious, but the resulting stabilization is independent of the order in which unstable vertices are fired. Thus, each configuration stabilizes to a unique stable configuration.

**Laplacian**

Fix a total order on the vertices of \( \Gamma \), thus leading to a labelling of the vertices by the numbers 1, 2, \ldots, \( n \) for some \( n \) (in the given order). The **Laplacian** of \( \Gamma \) is

\[
L := D - A
\]

where \( D \) is the diagonal matrix of out-degrees of the vertices (i.e., the diagonal \( n \times n \)-matrix whose \((i, i)\)-th entry is the out-degree of the vertex \( i \)) and \( A \) is the adjacency matrix whose \((i, j)\)-th entry is the weight of the edge from vertex \( i \) to vertex \( j \), which we take to be 0 if there is no edge. The **reduced Laplacian**, \( \tilde{L} \), is the submatrix of the Laplacian formed by removing the row and column corresponding to the sink vertex. Firing a vertex of a configuration is the same as subtracting the corresponding row of the reduced Laplacian.

**Example.** (Continued.)

```
sage: S.vertices(sort=True)  # the ordering of the vertices
[0, 1, 2, 3]
sage: S.laplacian()
[ 0 0 0 0]
[-1 4 -1 -2]
[ 0 -1 2 -1]
[ 0 -1 -1 2]
sage: S.reduced_laplacian()
[ 4 -1 -2]
[ -1 2 -1]
[ -1 -1 2]
```

The configuration we considered previously:
sage: c = SandpileConfig(S, [5,0,1])
sage: c
{1: 5, 2: 0, 3: 1}

Firing vertex 1 is the same as subtracting the corresponding row of the reduced Laplacian from the configuration (regarded as a vector):

sage: c.fire_vertex(1).values()
[1, 1, 3]
sage: S.reduced_laplacian()[0]
(4, -1, -2)
sage: vector([5,0,1]) - vector([4,-1,-2])
(1, 1, 3)

Recurrent elements

Imagine an experiment in which grains of sand are dropped one-at-a-time onto a graph, pausing to allow the configuration to stabilize between drops. Some configurations will only be seen once in this process. For example, for most graphs, once sand is dropped on the graph, no sequence of additions of sand and stabilizations will result in a graph empty of sand. Other configurations—the so-called recurrent configurations—will be seen infinitely often as the process is repeated indefinitely.

To be precise, a configuration $c$ is recurrent if (i) it is stable, and (ii) given any configuration $a$, there is a configuration $b$ such that $c = \text{stab}(a + b)$, the stabilization of $a + b$.

The maximal-stable configuration, denoted $c_{\text{max}}$, is defined by $c_{\text{max}}(v) = \text{out-degree}(v) - 1$ for all nonsink vertices $v$. It is clear that $c_{\text{max}}$ is recurrent. Further, it is not hard to see that a configuration is recurrent if and only if it has the form $\text{stab}(a + c_{\text{max}})$ for some configuration $a$.

Example. (Continued.)

sage: S.recurrents(verbos=false)
[[3, 1, 1], [2, 1, 1], [3, 1, 0]]
sage: c = SandpileConfig(S, [2,1,1])
sage: c
{1: 2, 2: 1, 3: 1}
sage: c.is_recurrent()
True

Adding any configuration to the max-stable configuration and stabilizing yields a recurrent configuration:

sage: x = SandpileConfig(S, [1,0,0])
sage: x + S.max_stable()
{1: 4, 2: 1, 3: 1}

Use & to add and stabilize:

sage: c = x & S.max_stable()
sage: c
{1: 3, 2: 1, 3: 0}
sage: c.is_recurrent()
True
Note the various ways of performing addition and stabilization:

```
sage: m = S.max_stable()
sage: (x + m).stabilize() == ~(x + m)
True
sage: (x + m).stabilize() == x & m
True
```

**Burning Configuration**

A *burning configuration* is a $\mathbb{Z}_{\geq 0}$-linear combination $f$ of the rows of the reduced Laplacian matrix having nonnegative entries and such that every vertex is accessible from some vertex in the support of $f$ (in other words, for each vertex $w$, there is a path from some $v \in \text{supp } f$ to $w$). The corresponding *burning script* gives the coefficients in the $\mathbb{Z}_{\geq 0}$-linear combination needed to obtain the burning configuration. So if $b$ is a burning configuration, $\sigma$ is its script, and $\tilde{L}$ is the reduced Laplacian, then $\sigma \tilde{L} = b$. The *minimal burning configuration* is the one with the minimal script (each of its components is no larger than the corresponding component of any other script for a burning configuration).

Given a burning configuration $b$ having script $\sigma$, and any configuration $c$ on the same graph, the following are equivalent:

- $c$ is recurrent;
- $c + b$ stabilizes to $c$;
- the firing vector for the stabilization of $c + b$ is $\sigma$.

The burning configuration and script are computed using a modified version of Speer’s script algorithm. This is a generalization to directed multigraphs of Dhar’s burning algorithm.

**Example.**

```
sage: g = {0:{},1:{0:1,3:1,4:1},2:{0:1,3:1,5:1},...
      3:{2:1,5:1},4:{1:1,3:1},5:{2:1,3:1}}
sage: G = Sandpile(g,0)
sage: G.burning_config()
{1: 2, 2: 0, 3: 1, 4: 1, 5: 0}
sage: G.burning_config().values()
[2, 0, 1, 1, 0]
sage: G.burning_script()
{1: 1, 2: 3, 3: 5, 4: 1, 5: 4}
sage: G.burning_script().values()
[1, 3, 5, 1, 4]
sage: matrix(G.burning_script().values())*G.reduced_laplacian()
[2 0 1 1 0]
```

**Sandpile group**

The collection of stable configurations forms a commutative monoid with addition defined as ordinary addition followed by stabilization. The identity element is the all-zero configuration. This monoid is a group exactly when the underlying graph is a DAG (directed acyclic graph).

The recurrent elements form a submonoid which turns out to be a group. This group is called the *sandpile group* for $\Gamma$, denoted $S(\Gamma)$. Its identity element is usually not the all-zero configuration (again, only in the case that $\Gamma$ is a DAG). So finding the identity element is an interesting problem.
Let $n = |V| - 1$ and fix an ordering of the nonsink vertices. Let $\tilde{L} \subset \mathbb{Z}^n$ denote the column-span of $\tilde{L}^t$, the transpose of the reduced Laplacian. It is a theorem that

$$S(\Gamma) \approx \mathbb{Z}^n / \tilde{L}. $$

Thus, the number of elements of the sandpile group is $\det \tilde{L}$, which by the matrix-tree theorem is the number of weighted trees directed into the sink.

**Example.** (Continued.)

```
sage: S.group_order()
3
sage: S.invariant_factors()
[1, 1, 3]
sage: S.reduced_laplacian().dense_matrix().smith_form()

([1 0 0]  [1 0 0]  [1 3 5]
 [0 1 0]  [0 1 0]  [1 4 6]
 [0 0 3],  [0 -1 1],  [1 4 7])
```

Adding the identity to any recurrent configuration and stabilizing yields the same recurrent configuration:

```
sage: S.identity()
{1: 3, 2: 1, 3: 0}
sage: i = S.identity()
sage: m = S.max_stable()
sage: i & m == m
True
```

### Self-organized criticality

The sandpile model was introduced by Bak, Tang, and Wiesenfeld in the paper, *Self-organized criticality: an explanation of $1/f$ noise* [BTW]. The term *self-organized criticality* has no precise definition, but can be loosely taken to describe a system that naturally evolves to a state that is barely stable and such that the instabilities are described by a power law. In practice, *self-organized criticality* is often taken to mean like the sandpile model on a grid graph. The grid graph is just a grid with an extra sink vertex. The vertices on the interior of each side have one edge to the sink, and the corner vertices have an edge of weight 2. Thus, every nonsink vertex has out-degree 4.

Imagine repeatedly dropping grains of sand on an empty grid graph, allowing the sandpile to stabilize in between. At first there is little activity, but as time goes on, the size and extent of the avalanche caused by a single grain of sand becomes hard to predict. Computer experiments—I do not think there is a proof, yet—indicate that the distribution of avalanche sizes obeys a power law with exponent -1. In the example below, the size of an avalanche is taken to be the sum of the number of times each vertex fires.

**Example (distribution of avalanche sizes).**

```
sage: S = sandpiles.Grid(10,10)
sage: m = S.max_stable()
sage: a = []
sage: for i in range(10000):  # long time (15s on sage.math, 2012)
...:    m = m.add_random()
...:    m, f = m.stabilize(true)
...:    a.append(sum(f.values()))
```

(continues on next page)
...  
sage: p = list_plot([[log(i+1),log(a.count(i))] for i in [0..max(a)] if a.count(i)])  
    # long time
sage: p.axes_labels(['log(N)','log(D(N))'])  
    # long time
sage: p  
    # long time
Graphics object consisting of 1 graphics primitive

Fig. 2: Distribution of avalanche sizes

Note: In the above code, m.stabilize(true) returns a list consisting of the stabilized configuration and the firing vector. (Omitting true would give just the stabilized configuration.)

Divisors and Discrete Riemann surfaces

A reference for this section is Riemann-Roch and Abel-Jacobi theory on a finite graph [BN].

A divisor on $\Gamma$ is an element of the free abelian group on its vertices, including the sink. Suppose, as above, that the $n + 1$ vertices of $\Gamma$ have been ordered, and that $L$ is the column span of the transpose of the Laplacian. A divisor is then identified with an element $D \in \mathbb{Z}^{n+1}$, and two divisors are linearly equivalent if they differ by an element of $L$. A divisor $E$ is effective, written $E \geq 0$, if $E(v) \geq 0$ for each $v \in V$, i.e., if $E \in \mathbb{Z}_{\geq 0}^{n+1}$. The degree of a divisor $D$ is $\text{deg}(D) := \sum_{v \in V} D(v)$. The divisors of degree zero modulo linear equivalence form the Picard group, or Jacobian of the graph. For an undirected graph, the Picard group is isomorphic to the sandpile group.

The complete linear system for a divisor $D$, denoted $|D|$, is the collection of effective divisors linearly equivalent to $D$.  

12.1. Thematic tutorial document tree
Riemann-Roch

To describe the Riemann-Roch theorem in this context, suppose that $\Gamma$ is an undirected, unweighted graph. The dimension, $r(D)$, of the linear system $|D|$ is $-1$ if $|D| = \emptyset$, and otherwise is the greatest integer $s$ such that $|D - E| \neq 0$ for all effective divisors $E$ of degree $s$. Define the canonical divisor by $K = \sum_{v \in V} (\deg(v) - 2)v$, and the genus by $g = \#(E) - \#(V) + 1$. The Riemann-Roch theorem says that for any divisor $D$,

$$r(D) - r(K - D) = \deg(D) + 1 - g.$$

Example.

```python
sage: G = sandpiles.Complete(5)  # the sandpile on the complete graph with 5 vertices

A divisor on the graph:
```
sage: D = SandpileDivisor(G, [1,2,2,0,2])
```

Verify the Riemann-Roch theorem:
```
sage: K = G.canonical_divisor()
sage: D.rank() - (K - D).rank() == D.deg() + 1 - G.genus()
```

```
True
```

The effective divisors linearly equivalent to $D$:
```
sage: D.effective_div(False)
```

```
[[0, 1, 1, 4, 1], [1, 2, 2, 0, 2], [4, 0, 0, 3, 0]]
```

The nonspecial divisors up to linear equivalence (divisors of degree $g - 1$ with empty linear systems):
```
sage: N = G.nonspecial_divisors()
sage: [E.values() for E in N[:5]]  # the first few
```

```
[-1, 0, 1, 2, 3],
[-1, 0, 1, 3, 2],
[-1, 0, 2, 1, 3],
[-1, 0, 2, 3, 1],
[-1, 0, 3, 1, 2]]
```

```
sage: len(N)
```

```
24
```

```
sage: len(N) == G.h_vector()[-1]
```

```
True
```

Picturing linear systems

Fix a divisor $D$. There are at least two natural graphs associated with the linear system associated with $D$. First, consider the directed graph with vertex set $|D|$ and with an edge from vertex $E$ to vertex $F$ if $F$ is attained from $E$ by firing a single unstable vertex.

```python
sage: S = Sandpile(graphs.CycleGraph(6),0)
sage: D = SandpileDivisor(S, [1,1,1,1,2,0])
sage: D.is_alive()
```

```
True
```

```
sage: eff = D.effective_div()
sage: firing_graph(S,eff).show3d(edge_size=0.005,vertex_size=0.01,iterations=500)
```

42 Chapter 12. Documentation
The is_alive method checks whether the divisor $D$ is alive, i.e., whether every divisor linearly equivalent to $D$ is unstable.

The second graph has the same set of vertices but with an edge from $E$ to $F$ if $F$ is obtained from $E$ by firing all unstable vertices of $E$.

```sage
sage: S = Sandpile(graphs.CycleGraph(6),0)
sage: D = SandpileDivisor(S, [1,1,1,1,2,0])
sage: eff = D.effective_div()
sage: parallel_firing_graph(S,eff).show3d(edge_size=.005,vertex_size=0.01,iterations=500)
```

Note that in each of the examples above, starting at any divisor in the linear system and following edges, one is eventually led into a cycle of length 6 (cycling the divisor $(1,1,1,1,2,0)$). Thus, D.alive() returns True. In Sage, one would be able to rotate the above figures to get a better idea of the structure.

### Algebraic geometry of sandpiles

#### Affine

Let $n = |V| - 1$, and fix an ordering on the nonsink vertices of $\Gamma$. Let $\hat{L} \subset \mathbb{Z}^n$ denote the column-span of $\hat{L}^t$, the transpose of the reduced Laplacian. Label vertex $i$ with the indeterminate $x_i$, and let $C[\Gamma_s] = C[x_1, \ldots, x_n]$. (Here, $s$ denotes the sink vertex of $\Gamma$.) The sandpile ideal or toppling ideal, first studied by Cori, Rossin, and Salvy [CRS] for undirected graphs, is the lattice ideal for $\hat{L}$:

$$I = I(\Gamma_s) := C[\Gamma_s] \cdot \{x^u - x^v : u - v \in \hat{L}\} \subset C[\Gamma_s],$$
where $x^u := \prod_{i=1}^n x^{u_i}$ for $u \in \mathbb{Z}^n$.

For each $c \in \mathbb{Z}^n$ define $t(c) = x^{c^+} - x^{c^-}$ where $c_i^+ = \max\{c_i, 0\}$ and $c_i^- = \max\{-c_i, 0\}$, so that $c = c^+ - c^-$. Then, for each $\sigma \in \mathbb{Z}^n$, define $T(\sigma) = t(L^\top \sigma)$. It then turns out that

$$I = (T(e_1), \ldots, T(e_n), x^b - 1),$$

where $e_i$ is the $i$-th standard basis vector and $b$ is any burning configuration.

The affine coordinate ring, $\mathbb{C}[\Gamma_s]/I$, is isomorphic to the group algebra of the sandpile group, $\mathbb{C}[\mathcal{S}(\Gamma)]$.

The standard term-ordering on $\mathbb{C}[\Gamma_s]$ is graded reverse lexicographical order with $x_i > x_j$ if vertex $v_i$ is further from the sink than vertex $v_j$. (There are choices to be made for vertices equidistant from the sink.) If $\sigma_b$ is the script for a burning configuration (not necessarily minimal), then

$$\{T(\sigma) : \sigma \leq \sigma_b\}$$

is a Gröbner basis for $I$. 

Fig. 4: Complete linear system for $(1, 1, 1, 2, 0)$ on $C_b$: parallel firings
Projective

Now let $C[\Gamma] = C[x_0, x_1, \ldots, x_n]$, where $x_0$ corresponds to the sink vertex. The \emph{homogeneous sandpile ideal}, denoted $I^h$, is obtaining by homogenizing $I$ with respect to $x_0$. Let $L$ be the (full) Laplacian, and $L \subseteq \mathbb{Z}^{n+1}$ be the column span of its transpose, $L^t$. Then $I^h$ is the lattice ideal for $L$:

$$I^h = I^h(\Gamma) := C[\Gamma] \cdot \{x^u - x^v : u - v \in L\} \subseteq C[\Gamma].$$

This ideal can be calculated by saturating the ideal

$$(T(e_i) : i = 0, \ldots, n)$$

with respect to the product of the indeterminates: $\prod_{i=0}^n x_i$ (extending the $T$ operator in the obvious way). A Gröbner basis with respect to the degree lexicographic order described above (with $x_0$ the smallest vertex) is obtained by homogenizing each element of the Gröbner basis for the non-homogeneous sandpile ideal with respect to $x_0$.

Example.

```
sage: g = {0:{}, 1:{0:1, 3:1, 4:1}, 2:{0:1, 3:1, 5:1}, ....: 3:{2:1, 5:1}, 4:{1:1, 3:1}, 5:{2:1, 3:1}}
sage: S = Sandpile(g, 0)
sage: S.ring()
Multivariate Polynomial Ring in x5, x4, x3, x2, x1, x0 over Rational Field
The homogeneous sandpile ideal:
```
```
sage: S.ideal()
Ideal (x2 - x0, x3^2 - x5*x0, x5*x3 - x0^2, x4^2 - x3*x1, x5^2 - x3*x0, x1^3 - x4*x3*x0, x4*x1^2 - x5*x0^2) of Multivariate Polynomial Ring in x5, x4, x3, x2, x1, x0 over Rational Field
The generators of the ideal:
```
```
sage: S.ideal(true)
[x2 - x0,
 x3^2 - x5*x0,
 x5*x3 - x0^2,
 x4^2 - x3*x1,
 x5^2 - x3*x0,
 x1^3 - x4*x3*x0,
 x4*x1^2 - x5*x0^2]
```

Its resolution:

```
sage: S.resolution() # long time
'R^1 <-- R^7 <-- R^19 <-- R^25 <-- R^16 <-- R^4'
```

and Betti table:

```
sage: S.betti() # long time

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1:</td>
<td>-</td>
<td>4</td>
<td>6</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>2:</td>
<td>-</td>
<td>2</td>
<td>7</td>
<td>7</td>
<td>2</td>
</tr>
</tbody>
</table>
```

(continues on next page)
The Hilbert function:

```
sage: S.hilbert_function()
[1, 5, 11, 15]
```

and its first differences (which count the number of superstable configurations in each degree):

```
sage: S.h_vector()
[1, 4, 6, 4]
```

```
sage: x = [i.deg() for i in S.superstables()]
sage: sorted(x)
[0, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3]
```

The degree in which the Hilbert function starts equalling the Hilbert polynomial, the latter always being a constant in the case of a sandpile ideal:

```
sage: S.postulation()
3
```

**Zeros**

The zero set for the sandpile ideal $I$ is

$$Z(I) = \{ p \in \mathbb{C}^n : f(p) = 0 \text{ for all } f \in I \},$$

the set of simultaneous zeros of the polynomials in $I$. Letting $S^1$ denote the unit circle in the complex plane, $Z(I)$ is a finite subgroup of $S^1 \times \cdots \times S^1 \subset \mathbb{C}^n$, isomorphic to the sandpile group. The zero set is actually linearly isomorphic to a faithful representation of the sandpile group on $\mathbb{C}^n$.

**Todo:** The above is not quite true. $Z(I)$ is neither finite nor a subgroup of $S^1 \times \cdots \times S^1 \subset \mathbb{C}^n$. What is probably meant is that the subset of $Z(I)$ in which the coordinate of $p$ corresponding to the sink is set to 1 is a finite subgroup of $S^1 \times \cdots \times S^1 \subset \mathbb{C}^{n-1}$.

**Example.** (Continued.)

```
sage: S = Sandpile({0: {}, 1: {2: 2}, 2: {0: 4, 1: 1}}, 0)
sage: S.ideal().gens()
x1^2 - x2^2, x1*x2^3 - x0^4, x2^5 - x1*x0^4
```

Approximation to the zero set (setting $x_0 = 1$):

```
sage: S.solve()
[[-0.707107000000000 + 0.707107000000000*I, 0.707107000000000 - 0.707107000000000*I],
 [-0.707107000000000 - 0.707107000000000*I, 0.707107000000000 + 0.707107000000000*I],
 [-I, -I]]
```

(continues on next page)
The zeros are generated as a group by a single vector:

\[
\text{sage: } \text{S.points()}
\]
\[
[\-(1/2*I + 1/2)*\sqrt(2), (1/2*I + 1/2)*\sqrt(2)]
\]

Resolutions

The homogeneous sandpile ideal, \( I^h \), has a free resolution graded by the divisors on \( \Gamma \) modulo linear equivalence. (See the section on Discrete Riemann Surfaces for the language of divisors and linear equivalence.) Let \( S = C[\Gamma] = C[x_0, \ldots, x_n] \), as above, and let \( \mathfrak{S} \) denote the group of divisors modulo rational equivalence. Then \( S \) is graded by \( \mathfrak{S} \) by letting \( \deg(x^c) = c \in \mathfrak{S} \) for each monomial \( x^c \). The minimal free resolution of \( I^h \) has the form

\[
0 \leftarrow I^h \leftarrow \bigoplus_{D \in \mathfrak{S}} S(-D)^{\beta_{0,D}} \leftarrow \bigoplus_{D \in \mathfrak{S}} S(-D)^{\beta_{1,D}} \leftarrow \cdots \leftarrow \bigoplus_{D \in \mathfrak{S}} S(-D)^{\beta_{r,D}} \leftarrow 0,
\]

where the \( \beta_{i,D} \) are the Betti numbers for \( I^h \).

For each divisor class \( D \in \mathfrak{S} \), define a simplicial complex

\[
\Delta_D := \{ I \subseteq \{0, \ldots, n\} : I \subseteq \text{supp}(E) \text{ for some } E \in |D| \}.
\]

The Betti number \( \beta_{i,D} \) equals the dimension over \( C \) of the \( i \)-th reduced homology group of \( \Delta_D \):

\[
\beta_{i,D} = \dim_C \tilde{H}_i(\Delta_D; C).
\]

Representatives of all divisor classes with nontrivial homology:

\[
\text{sage: } S = \text{Sandpile}([0: {}], [1: {0: 1, 2: 1, 3: 4}, 2: {3: 5}, 3: {1: 1, 2: 1}], 0)
\]

The homology associated with the first divisor in the list:

\[
\text{sage: } D = p[0][0]
\]
\[
\text{sage: } D.\text{effective_div()}
\]
\[
[0: 0, 1: 0, 2: 0, 3: 2], [0: 0, 1: 1, 2: 1, 3: 0]
\]

\[
\text{sage: } [E.\text{support()} \text{ for } E \text{ in } D.\text{effective_div()}]
\]
\[
[[3], [1, 2]]
\]
The minimal free resolution:

```
sage: S.resolution()
'\mathcal{R}^1 \leftarrow \mathcal{R}^5 \leftarrow \mathcal{R}^5 \leftarrow \mathcal{R}^1'
sage: S.betti()

\begin{array}{cccc}
0 & 1 & 2 & 3 \\
\hline
0: & 1 & - & - & - \\
1: & - & 5 & 5 & - \\
2: & - & - & - & 1 \\
\hline
\text{total:} & 1 & 5 & 5 & 1
\end{array}
```

The degrees and ranks of the homology groups for each element of the list `p` (compare with the Betti table, above):

```
sage: [[sum(d[0].values()),d[1].betti()] for d in p]
[[2, {0: 2, 1: 0}],
 [3, {0: 1, 1: 1, 2: 0}],
 [2, {0: 2, 1: 0}],
 [3, {0: 1, 1: 1, 2: 0}],
 [2, {0: 2, 1: 0}],
 [3, {0: 1, 1: 1}],
 [2, {0: 2, 1: 0}],
 [3, {0: 1, 1: 1, 2: 0}],
 [5, {0: 1, 1: 0, 2: 1}]]
```

**Complete Intersections and Arithmetically Gorenstein toppling ideals**

NOTE: in the previous section note that the resolution always has length `n` since the ideal is Cohen-Macaulay.

To do.

**Betti numbers for undirected graphs**

To do.
Usage

Initialization

There are three main classes for sandpile structures in Sage: \texttt{Sandpile}, \texttt{SandpileConfig}, and \texttt{SandpileDivisor}. Initialization for \texttt{Sandpile} has the form

\begin{verbatim}
.. skip

sage: S = Sandpile(graph, sink)
\end{verbatim}

where \texttt{graph} represents a graph and \texttt{sink} is the key for the sink vertex. There are four possible forms for \texttt{graph}:

1. a Python dictionary of dictionaries:

\begin{verbatim}
sage: g = {0: {}, 1: {0: 1, 3: 1, 4: 1}, 2: {0: 1, 3: 1, 5: 1}, ....: 3: {2: 1, 5: 1}, 4: {1: 1, 3: 1}, 5: {2: 1, 3: 1}}
\end{verbatim}

Graph from dictionary of dictionaries.

Each key is the name of a vertex. Next to each vertex name $v$ is a dictionary consisting of pairs: \texttt{vertex: weight}. Each pair represents a directed edge emanating from $v$ and ending at $\texttt{vertex}$ having (non-negative integer) weight equal to \texttt{weight}. Loops are allowed. In the example above, all of the weights are 1.

2. a Python dictionary of lists:

\begin{verbatim}
sage: g = {0: [], 1: [0, 3, 4], 2: [0, 3, 5], ....: 3: [2, 5], 4: [1, 3], 5: [2, 3]}
\end{verbatim}

This is a short-hand when all of the edge-weights are equal to 1. The above example is for the same displayed graph.
3. a Sage graph (of type `sage.graphs.graph.Graph`):

\[
\begin{align*}
\texttt{sage: } & \quad \texttt{g = graphs.CycleGraph(5)} \\
\texttt{sage: } & \quad \texttt{S = Sandpile(g, 0)} \\
\texttt{sage: } & \quad \texttt{type(g)} \\
& \quad \texttt{<class 'sage.graphs.graph.Graph'>}
\end{align*}
\]

To see the types of built-in graphs, type `graphs.`, including the period, and hit TAB.

4. a Sage digraph:

\[
\begin{align*}
\texttt{sage: } & \quad \texttt{S = Sandpile(digraphs.RandomDirectedGNC(6), 0)} \\
\texttt{sage: } & \quad \texttt{S.show()}
\end{align*}
\]

![Fig. 5: A random graph.](image)

See `sage.graphs.graph_generators` for more information on the Sage graph library and graph constructors.

Each of these four formats is preprocessed by the Sandpile class so that, internally, the graph is represented by the dictionary of dictionaries format first presented. This internal format is returned by `dict()`:

\[
\begin{align*}
\texttt{sage: } & \quad \texttt{S = Sandpile({0: [], 1: [0, 3, 4], 2: [0, 3, 5], 3: [2, 5], 4: [1, 3], 5: [2, 3]}, 0)} \\
\texttt{sage: } & \quad \texttt{S.dict()} \\
& \quad \texttt{{0: {}, 1: {0: 1, 3: 1, 4: 1}, 2: {0: 1, 3: 1, 5: 1}, 3: {2: 1, 5: 1}, 4: {1: 1, 3: 1}, 5: {2: 1, 3: 1}}} \\
\end{align*}
\]
**Note:** The user is responsible for assuring that each vertex has a directed path into the designated sink. If the sink has out-edges, these will be ignored for the purposes of sandpile calculations (but not calculations on divisors).

Code for checking whether a given vertex is a sink:

```sage
sage: S = Sandpile({0:[], 1:[0, 3, 4], 2:[0, 3, 5], 3: [2, 5], 4: [1, 3], 5: [2, 3]},0)
sage: [S.distance(v,0) for v in S.vertices(sort=True)] # 0 is a sink
[0, 1, 1, 2, 2, 2]
sage: [S.distance(v,1) for v in S.vertices(sort=True)] # 1 is not a sink
[+Infinity, 0, +Infinity, +Infinity, 1, +Infinity]
```

**Methods**

Here are summaries of Sandpile, SandpileConfig, and SandpileDivisor methods (functions). Each summary is followed by a list of complete descriptions of the methods. There are many more methods available for a Sandpile, e.g., those inherited from the class DiGraph. To see them all, enter `dir(Sandpile)` or type `Sandpile`, including the period, and hit TAB.

**Sandpile**

**Summary of methods.**

- `all_k_config` — The constant configuration with all values set to k.
- `all_k_div` — The divisor with all values set to k.
- `avalanche_polynomial` — The avalanche polynomial.
- `betti` — The Betti table for the homogeneous toppling ideal.
- `betti_complexes` — The support-complexes with non-trivial homology.
- `burning_config` — The minimal burning configuration.
- `burning_script` — A script for the minimal burning configuration.
- `canonical_divisor` — The canonical divisor.
- `dict` — A dictionary of dictionaries representing a directed graph.
- `genus` — The genus: (# non-loop edges) - (# vertices) + 1.
- `groebner` — A Groebner basis for the homogeneous toppling ideal.
- `group_gens` — A minimal list of generators for the sandpile group.
- `group_order` — The size of the sandpile group.
- `h_vector` — The number of superstable configurations in each degree.
- `help` — List of Sandpile-specific methods (not inherited from Graph).
- `hilbert_function` — The Hilbert function of the homogeneous toppling ideal.
- `ideal` — The saturated homogeneous toppling ideal.
- `identity` — The identity configuration.
- `in_degree` — The in-degree of a vertex or a list of all in-degrees.
• invariant_factors — The invariant factors of the sandpile group.
• is_undirected — Is the underlying graph undirected?
• jacobian_representatives — Representatives for the elements of the Jacobian group.
• laplacian — The Laplacian matrix of the graph.
• markov_chain — The sandpile Markov chain for configurations or divisors.
• max_stable — The maximal stable configuration.
• max_stable_div — The maximal stable divisor.
• max_superstables — The maximal superstable configurations.
• min_recurrences — The minimal recurrent elements.
• nonsink_vertices — The nonsink vertices.
• nonspecial_divisors — The nonspecial divisors.
• out_degree — The out-degree of a vertex or a list of all out-degrees.
• picard_representatives — Representatives of the divisor classes of degree d in the Picard group.
• points — Generators for the multiplicative group of zeros of the sandpile ideal.
• postulation — The postulation number of the toppling ideal.
• recurrences — The recurrent configurations.
• reduced_laplacian — The reduced Laplacian matrix of the graph.
• reorder_vertices — A copy of the sandpile with vertex names permuted.
• resolution — A minimal free resolution of the homogeneous toppling ideal.
• ring — The ring containing the homogeneous toppling ideal.
• show — Draw the underlying graph.
• show3d — Draw the underlying graph.
• sink — The sink vertex.
• smith_form — The Smith normal form for the Laplacian.
• solve — Approximations of the complex affine zeros of the sandpile ideal.
• stable_configs — Generator for all stable configurations.
• stationary_density — The stationary density of the sandpile.
• superstables — The superstable configurations.
• symmetric_recurrences — The symmetric recurrent configurations.
• tutte_polynomial — The Tutte polynomial.
• unsaturated_ideal — The unsaturated, homogeneous toppling ideal.
• version — The version number of Sage Sandpiles.
• zero_config — The all-zero configuration.
• zero_div — The all-zero divisor.
Complete descriptions of Sandpile methods.

all_k_config(k)
The constant configuration with all values set to $k$.

**INPUT:**
- k – integer

**OUTPUT:**
- SandpileConfig

**EXAMPLES:**

```python
sage: s = sandpiles.Diamond()
sage: s.all_k_config(7)
{1: 7, 2: 7, 3: 7}
```

all_k_div(k)
The divisor with all values set to $k$.

**INPUT:**
- k – integer

**OUTPUT:**
- SandpileDivisor

**EXAMPLES:**

```python
sage: S = sandpiles.House()
sage: S.all_k_div(7)
{0: 7, 1: 7, 2: 7, 3: 7, 4: 7}
```

avalanche_polynomial(multivariable=True)
The avalanche polynomial. See NOTE for details.

**INPUT:**
- multivariable – (default: True) boolean

**OUTPUT:**
- polynomial

**EXAMPLES:**

```python
sage: s = sandpiles.Complete(4)
sage: s.avalanche_polynomial()
9*x0*x1*x2 + 2*x0*x1 + 2*x0*x2 + 2*x1*x2 + 3*x0 + 3*x1 + 3*x2 + 24
sage: s.avalanche_polynomial(False)
9*x0^3 + 6*x0^2 + 9*x0 + 24
```
Note: For each nonsink vertex \( v \), let \( x_v \) be an indeterminate. If \((r, v)\) is a pair consisting of a recurrent \( r \) and nonsink vertex \( v \), then for each nonsink vertex \( w \), let \( n_w \) be the number of times vertex \( w \) fires in the stabilization of \( r + v \). Let \( M(r, v) \) be the monomial \( \prod_w x_w^{n_w} \), i.e., the exponent records the vector of \( n_w \) as \( w \) ranges over the nonsink vertices. The avalanche polynomial is then the sum of \( M(r, v) \) as \( r \) ranges over the recurrents and \( v \) ranges over the nonsink vertices. If \texttt{multivariable} is \texttt{False}, then set all the indeterminates equal to each other (and, thus, only count the number of vertex firings in the stabilizations, forgetting which particular vertices fired).

---

\texttt{betti(\texttt{verbose=\texttt{True}})}

The Betti table for the homogeneous toppling ideal. If \texttt{verbose} is \texttt{True}, it prints the standard Betti table, otherwise, it returns a less formatted table.

**INPUT:**

\texttt{verbose} – (default: \texttt{True}) boolean

**OUTPUT:**

Betti numbers for the sandpile

**EXAMPLES:**

```sage
sage: S = sandpiles.Diamond()
sage: S.betti()

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>4</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>total:</td>
<td>1</td>
<td>6</td>
<td>9</td>
<td>4</td>
</tr>
</tbody>
</table>
```

```sage
sage: S.betti(False)
[1, 6, 9, 4]
```

---

\texttt{betti\_complexes()}

The support-complexes with non-trivial homology. (See NOTE.)

**OUTPUT:**

list (of pairs [divisors, corresponding simplicial complex])

**EXAMPLES:**

```sage
sage: S = Sandpile({0:{},1:{0: 1, 2: 1, 3: 4},2:{3: 5},3:{1: 1, 2: 1}},0)
sage: p = S.betti_complexes()
sage: p[0]
[{0: -8, 1: 5, 2: 4, 3: 1}, Simplicial complex with vertex set (1, 2, 3) and facets {{3, --}, (1, 2)}]
sage: S.resolution()
'R^1 <-- R^5 <-- R^5 <-- R^1'
sage: S.betti()

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
</table>
```

(continues on next page)
Note: A support–complex is the simplicial complex formed from the supports of the divisors in a linear system.

—

`burning_config()`

The minimal burning configuration.

OUTPUT:

dict (configuration)

EXAMPLES:

```
sage: g = {0:{},1:{0:1,3:1,4:1},2:{0:1,3:1,5:1},
       3:{2:1,5:1},4:{1:1,3:1},5:{2:1,3:1}}
sage: S = Sandpile(g,0)
sage: S.burning_config()
{1: 2, 2: 0, 3: 1, 4: 1, 5: 0}
sage: S.burning_config().values()
[2, 0, 1, 1, 0]
sage: S.burning_script()
{1: 1, 2: 3, 3: 5, 4: 1, 5: 4}
sage: script = S.burning_script().values()
sage: script
[1, 3, 5, 1, 4]
sage: matrix(script)*S.reduced_laplacian()
[2 0 1 1 0]
```

Note: The burning configuration and script are computed using a modified version of Speer’s script algorithm. This is a generalization to directed multigraphs of Dhar’s burning algorithm.

A burning configuration is a nonnegative integer-linear combination of the rows of the reduced Laplacian matrix having nonnegative entries and such that every vertex has a path from some vertex in its support. The corresponding burning script gives the integer-linear combination needed to obtain the burning configuration. So if $b$ is the burning configuration, $\sigma$ is its script, and $\tilde{L}$ is the reduced Laplacian, then $\sigma \cdot \tilde{L} = b$. The minimal burning configuration is the one with the minimal script (its components are no larger than the components of any other script for a burning configuration).

The following are equivalent for a configuration $c$ with burning configuration $b$ having script $\sigma$:

- $c$ is recurrent;
- $c + b$ stabilizes to $c$;

12.1. Thematic tutorial document tree
• the firing vector for the stabilization of \( c + b \) is \( \sigma \).

---

**burning_script()**

A script for the minimal burning configuration.

**OUTPUT:**

dict

**EXAMPLES:**

```python
sage: g = {0:{}, 1:{0:1, 3:1, 4:1}, 2:{0:1, 3:1, 5:1},
       3:{2:1, 5:1}, 4:{1:1, 3:1}, 5:{2:1, 3:1}}
sage: S = Sandpile(g, 0)
sage: S.burning_config()
{1: 2, 2: 0, 3: 1, 4: 1, 5: 0}
sage: S.burning_config().values()
[2, 0, 1, 1, 0]
sage: S.burning_script()
{1: 1, 2: 3, 3: 5, 4: 1, 5: 4}
sage: script = S.burning_script().values()
sage: script
[1, 3, 5, 1, 4]
sage: matrix(script)*S.reduced_laplacian()
[2 0 1 1 0]
```

**Note:** The burning configuration and script are computed using a modified version of Speer’s script algorithm. This is a generalization to directed multigraphs of Dhar’s burning algorithm.

A **burning configuration** is a nonnegative integer-linear combination of the rows of the reduced Laplacian matrix having nonnegative entries and such that every vertex has a path from some vertex in its support. The corresponding **burning script** gives the integer-linear combination needed to obtain the burning configuration. So if \( b \) is the burning configuration, \( s \) is its script, and \( L_{\text{red}} \) is the reduced Laplacian, then \( s \cdot L_{\text{red}} = b \). The **minimal burning configuration** is the one with the minimal script (its components are no larger than the components of any other script for a burning configuration).

The following are equivalent for a configuration \( c \) with burning configuration \( b \) having script \( s \):

• \( c \) is recurrent;
• \( c + b \) stabilizes to \( c \);
• the firing vector for the stabilization of \( c + b \) is \( s \).

---

**canonical_divisor()**

The canonical divisor. This is the divisor with \( \text{deg}(v) - 2 \) grains of sand on each vertex (not counting loops). Only for undirected graphs.

**OUTPUT:**

SandpileDivisor

**EXAMPLES:**
sage: S = sandpiles.Complete(4)
sage: S.canonical_divisor()
{0: 1, 1: 1, 2: 1, 3: 1}
sage: s = Sandpile({0:[1,1],1:[0,0,1,1]},0)
sage: s.canonical_divisor() # loops are disregarded
{0: 0, 1: 0}

Warning: The underlying graph must be undirected.

—

dict()
A dictionary of dictionaries representing a directed graph.

OUTPUT:
dict

EXAMPLES:

sage: S = sandpiles.Diamond()
sage: S.dict()
{0: {1: 1, 2: 1},
  1: {0: 1, 2: 1, 3: 1},
  2: {0: 1, 1: 1, 3: 1},
  3: {1: 1, 2: 1}}
sage: S.sink()
0

—

genus()
The genus: (# non-loop edges) - (# vertices) + 1. Only defined for undirected graphs.

OUTPUT:
integer

EXAMPLES:

sage: sandpiles.Complete(4).genus()
3
sage: sandpiles.Cycle(5).genus()
1

—

groebner()
A Groebner basis for the homogeneous toppling ideal. It is computed with respect to the standard sandpile ordering (see ring).

OUTPUT:
Groebner basis

EXAMPLES:
sage: S = sandpiles.Diamond()
sage: S.groebner()
[x^3*x^2 - x1^2*x0, x2^3 - x3*x1*x0, x3^2 - x0^2, x2*x1 - x0^2, x^3 - x2^2*x0, x1^3 - x3*x2*x0, x3^2 - x0^2, x2^2*x0, x1^1 - x3^1, x2^1 - x1^1, x0^1 - x3^1, x1^2 - x0^2, x2^2 - x1^2, x0^2 - x2^2, x1^1 - x0^1, x2^1 - x1^1, x0^1 - x2^1, x1^0 - x0^0, x2^0 - x1^0, x0^0 - x2^0]

—

**group_gens(verbose=True)**

A minimal list of generators for the sandpile group. If `verbose` is `False` then the generators are represented as lists of integers.

**INPUT:**

`verbose` – (default: `True`) boolean

**OUTPUT:**

list of SandpileConfig (or of lists of integers if `verbose` is `False`)

**EXAMPLES:**

```python
ts = sandpiles.Cycle(5)
ts.group_gens()  # Output: [{1: 0, 2: 1, 3: 1, 4: 1}]
ts.group_gens()[0].order()  # Output: 5
	s = sandpiles.Complete(5)
ts.group_gens(False)  # Output: [[2, 3, 2, 2], [2, 2, 3, 2], [2, 2, 2, 3]]
[t.order() for t in ts.group_gens()]  # Output: [5, 5, 5]
ts.invariant_factors()  # Output: [1, 5, 5, 5]
```

—

**group_order()**

The size of the sandpile group.

**OUTPUT:**

integer

**EXAMPLES:**

```python
S = sandpiles.House()
S.group_order()  # Output: 11
```

—

**h_vector()**

The number of superstable configurations in each degree. Equivalently, this is the list of first differences of the Hilbert function of the (homogeneous) toppling ideal.

**OUTPUT:**

list of nonnegative integers
EXAMPLES:

```
sage: s = sandpiles.Grid(2,2)
sage: s.hilbert_function()
[1, 5, 15, 35, 66, 106, 146, 178, 192]
sage: s.h_vector()
[1, 4, 10, 20, 31, 40, 40, 32, 14]
```

---

**help(\texttt{verbose=True})**

List of Sandpile-specific methods (not inherited from Graph). If \texttt{verbose}, include short descriptions.

**INPUT:**

\texttt{verbose} – (default: \texttt{True}) boolean

**OUTPUT:**

printed string

**EXAMPLES:**

```
sage: Sandpile.help()
For detailed help with any method \texttt{FOO} listed below, enter "Sandpile.FOO?" or enter "S.FOO?" for any Sandpile \texttt{S}.

\texttt{all\_k\_config} -- The constant configuration with all values set to \texttt{k}.
\texttt{all\_k\_div} -- The divisor with all values set to \texttt{k}.
\texttt{avalanche\_polynomial} -- The avalanche polynomial.
\texttt{betti} -- The Betti table for the homogeneous toppling ideal.
\texttt{betti\_complexes} -- The support-complexes with non-trivial homology.
...\texttt{unsaturated\_ideal} -- The unsaturated, homogeneous toppling ideal.
\texttt{version} -- The version number of Sage Sandpiles.
\texttt{zero\_config} -- The all-zero configuration.
\texttt{zero\_div} -- The all-zero divisor.
```

---

**hilbert\_function()**

The Hilbert function of the homogeneous toppling ideal.

**OUTPUT:**

list of nonnegative integers

**EXAMPLES:**

```
sage: s = sandpiles.Wheel(5)
sage: s.hilbert_function()
[1, 5, 15, 31, 45]
sage: s.h_vector()
[1, 4, 10, 16, 14]
```

---

**ideal(gens=False)**

The saturated homogeneous toppling ideal. If \texttt{gens} is \texttt{True}, the generators for the ideal are returned instead.

12.1. Thematic tutorial document tree
**Thematic Tutorials, Release 9.7**

**INPUT:**

gens – (default: False) boolean

**OUTPUT:**

ideal or, optionally, the generators of an ideal

**EXAMPLES:**

```python
sage: S = sandpiles.Diamond()
sage: S.ideal()
Ideal (x2*x1 - x0^2, x3^2 - x0^2, x1^3 - x3*x2*x0, x3*x1^2 - x2^2*x0, x2^3 - x3*x1*x0, x3*x2^2 - x1^2*x0) of Multivariate Polynomial Ring in x3, x2, x1, x0 over Rational Field
sage: S.ideal(True)
[x2*x1 - x0^2, x3^2 - x0^2, x1^3 - x3*x2*x0, x3*x1^2 - x2^2*x0, x2^3 - x3*x1*x0, x3*x2^2 - x1^2*x0]
sage: S.ideal().gens()  # another way to get the generators
[x2*x1 - x0^2, x3^2 - x0^2, x1^3 - x3*x2*x0, x3*x1^2 - x2^2*x0, x2^3 - x3*x1*x0, x3*x2^2 - x1^2*x0]
```

**identity(verbose=True)**

The identity configuration. If `verbose` is `False`, the configuration are converted to a list of integers.

**INPUT:**

verbose – (default: True) boolean

**OUTPUT:**

SandpileConfig or a list of integers If `verbose` is `False`, the configuration are converted to a list of integers.

**EXAMPLES:**

```python
sage: s = sandpiles.Diamond()
sage: s.identity()
{1: 2, 2: 2, 3: 0}
sage: s.identity(False)
[2, 2, 0]
sage: s.identity() & s.max_stable() == s.max_stable()
True
```

**in_degree(v=None)**

The in-degree of a vertex or a list of all in-degrees.

**INPUT:**

v – (optional) vertex name

**OUTPUT:**

integer or dict

**EXAMPLES:**
```python
sage: s = sandpiles.House()
sage: s.in_degree()
{0: 2, 1: 2, 2: 3, 3: 3, 4: 2}
sage: s.in_degree(2)
3
```

### invariant_factors()

The invariant factors of the sandpile group.

**OUTPUT:**

list of integers

**EXAMPLES:**

```python
sage: s = sandpiles.Grid(2,2)
sage: s.invariant_factors()
[1, 1, 8, 24]
```

### is_undirected()

Is the underlying graph undirected? True if \((u, v)\) is and edge if and only if \((v, u)\) is an edge, each edge with the same weight.

**OUTPUT:**

boolean

**EXAMPLES:**

```python
sage: sandpiles.Complete(4).is_undirected()
True
sage: s = Sandpile({0:[1,2], 1:[0,2], 2:[0]}, 0)
sage: s.is_undirected()
False
```

### jacobian_representatives(\texttt{verbose=\texttt{True}})

Representatives for the elements of the Jacobian group. If \texttt{verbose} is \texttt{False}, then lists representing the divisors are returned.

**INPUT:**

\texttt{verbose} – (default: \texttt{True}) boolean

**OUTPUT:**

list of SandpileDivisor (or of lists representing divisors)

**EXAMPLES:**

For an undirected graph, divisors of the form \(s - \deg(s) \cdot \text{sink}\) as \(s\) varies over the superstables forms a distinct set of representatives for the Jacobian group:
If the graph is directed, the representatives described above may be equivalent modulo the rowspan of the Laplacian matrix:

```sage
s = Sandpile({0: {1: 1, 2: 2}, 1: {0: 2, 2: 4}, 2: {0: 4, 1: 2}},0)
sage: s.group_order()
28
sage: s.jacobian_reps()
[{-5, 3, 2}, {-4, 3, 1}]
```

Let $\tau$ be the nonnegative generator of the kernel of the transpose of the Laplacian, and let $\tau_s$ be its sink component, then the sandpile group is isomorphic to the direct sum of the cyclic group of order $\tau_s$ and the Jacobian group. In the example above, we have:

```sage
sage: s.laplacian().left_kernel()
Free module of degree 3 and rank 1 over Integer Ring
Echelon basis matrix:
[14 5 8]
```

**Note:** The Jacobian group is the set of all divisors of degree zero modulo the integer rowspan of the Laplacian matrix.

---

### laplacian()

The Laplacian matrix of the graph. Its rows encode the vertex firing rules.

**OUTPUT:**

matrix

**EXAMPLES:**

```sage
sage: G = sandpiles.Diamond()
sage: G.laplacian()
[ 2 -1 -1 -1]
[-1 3 -1 -1]
[-1 -1 3 -1]
[ 0 -1 -1 2]
```

**Warning:** The function `laplacian_matrix` should be avoided. It returns the indegree version of the Laplacian.

---

### markov_chain(state, distr=None)

The sandpile Markov chain for configurations or divisors. The chain starts at `state`. See NOTE for details.

**INPUT:**
• **state** – SandpileConfig, SandpileDivisor, or list representing one of these

• **distrib** – (optional) list of nonnegative numbers summing to 1 (representing a prob. dist.)

**OUTPUT:**

generator for Markov chain (see NOTE)

**EXAMPLES:**

```
sage: s = sandpiles.Complete(4)
sage: m = s.markov_chain([0,0,0])
sage: next(m)                            # random
{1: 0, 2: 0, 3: 0}
sage: next(m).values()                   # random
[0, 0, 0]
sage: next(m).values()                   # random
[0, 0, 0]
sage: next(m).values()                   # random
[0, 0, 0]
sage: next(m).values()                   # random
[0, 1, 0]
sage: next(m).values()                   # random
[0, 2, 0]
sage: next(m).values()                   # random
[0, 2, 1]
sage: next(m).values()                   # random
[1, 2, 1]
sage: next(m).values()                   # random
[2, 2, 1]
sage: m = s.markov_chain(s.zero_div(), [0.1,0.1,0.1,0.7])
sage: next(m).values()                   # random
[0, 0, 0, 1]
sage: next(m).values()                   # random
[0, 0, 1, 1]
sage: next(m).values()                   # random
[0, 1, 1, 2]
sage: next(m).values()                   # random
[1, 1, 2, 0]
sage: next(m).values()                   # random
[1, 1, 2, 1]
sage: next(m).values()                   # random
[1, 1, 2, 2]
sage: next(m).values()                   # random
[1, 1, 2, 3]
sage: next(m).values()                   # random
[1, 1, 2, 4]
sage: next(m).values()                   # random
[1, 1, 3, 4]
```

**Note:** The closed sandpile Markov chain has state space consisting of the configurations on a sandpile. It transitions from a state by choosing a vertex at random (according to the probability distribution `distrib`), dropping a grain of sand at that vertex, and stabilizing. If the chosen vertex is the sink, the chain stays at the current state.

The open sandpile Markov chain has state space consisting of the recurrent elements, i.e., the state space is the sandpile group. It transitions from the configuration `c` by choosing a vertex `v` at random according to `distrib`. The
next state is the stabilization of $c + v$. If $v$ is the sink vertex, then the stabilization of $c + v$ is defined to be $c$.

Note that in either case, if `distrib` is specified, its length is equal to the total number of vertices (including the sink).

REFERENCES:
—

**max_stable()**

The maximal stable configuration.

OUTPUT:

SandpileConfig (the maximal stable configuration)

EXAMPLES:

```sage
sage: S = sandpiles.House()
sage: S.max_stable()
{1: 1, 2: 2, 3: 2, 4: 1}
```

—

**max_stable_div()**

The maximal stable divisor.

OUTPUT:

SandpileDivisor (the maximal stable divisor)

EXAMPLES:

```sage
sage: s = sandpiles.Diamond()
sage: s.max_stable_div()
{0: 1, 1: 2, 2: 2, 3: 1}
sage: s.out_degree()
{0: 2, 1: 3, 2: 3, 3: 2}
```

—

**max_superstables(verbos=True)**

The maximal superstable configurations. If the underlying graph is undirected, these are the superstables of highest degree. If `verbos` is `False`, the configurations are converted to lists of integers.

INPUT:

`verbos` – (default: `True`) boolean

OUTPUT:

tuple of SandpileConfig

EXAMPLES:

```sage
sage: s = sandpiles.Diamond()
sage: s.superstables(False)
[[0, 0, 0],
 [0, 0, 1],
 [1, 0, 1],
 [0, 2, 0]],
```
min_recurrents(\texttt{verbose=\texttt{True}})

The minimal recurrent elements. If the underlying graph is undirected, these are the recurrent elements of least degree. If \texttt{verbose} is \texttt{False}, the configurations are converted to lists of integers.

INPUT:

\texttt{verbose} – (default: \texttt{True}) boolean

OUTPUT:

list of \texttt{SandpileConfig}

EXAMPLES:

\begin{verbatim}
sage: s = sandpiles.Diamond()
sage: s.recurrents(\texttt{False})
[[2, 2, 1],
 [2, 2, 0],
 [1, 2, 0],
 [2, 0, 1],
 [0, 2, 1],
 [1, 2, 1],
 [2, 1, 1]]
sage: s.min_recurrents(\texttt{False})
[[1, 2, 0], [2, 0, 1], [0, 2, 1], [2, 1, 0]]
sage: [i.deg() for i in s.recurrents()]
[5, 4, 3, 3, 3, 3, 4, 4]
\end{verbatim}

–––

\texttt{nonsink_vertices()}

The nonsink vertices.

OUTPUT:

list of vertices

EXAMPLES:

\begin{verbatim}
sage: s = sandpiles.Grid(2,3)
sage: s.nonsink_vertices()
[(1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (2, 3)]
\end{verbatim}

–––

\texttt{nonspecial_divisors(\texttt{verbose=\texttt{True}})
The nonspecial divisors. Only for undirected graphs. (See NOTE.)

**INPUT:**

verbose – (default: True) boolean

**OUTPUT:**

list (of divisors)

**EXAMPLES:**

```python
sage: S = sandpiles.Complete(4)
sage: ns = S.nonspecial_divisors()
sage: D = ns[0]
sage: D.values()
[-1, 0, 1, 2]
sage: D.deg()
2
sage: [i.effective_div() for i in ns]
[], [], [], [], [], []
```

**Note:** The “nonspecial divisors” are those divisors of degree \( g - 1 \) with empty linear system. The term is only defined for undirected graphs. Here, \( g = |E| - |V| + 1 \) is the genus of the graph (not counting loops as part of \( |E| \)). If `verbose` is False, the divisors are converted to lists of integers.

**Warning:** The underlying graph must be undirected.

---

**out_degree(v=None)**

The out-degree of a vertex or a list of all out-degrees.

**INPUT:**

`v` - (optional) vertex name

**OUTPUT:**

integer or dict

**EXAMPLES:**

```python
sage: s = sandpiles.House()
sage: s.out_degree()
{0: 2, 1: 2, 2: 3, 3: 3, 4: 2}
sage: s.out_degree(2)
3
```

---

**picard_representatives(d, verbose=True)**

Representatives of the divisor classes of degree \( d \) in the Picard group. (Also see the documentation for `jacobian_representatives`.)

**INPUT:**
• `d` – integer
  • `verbose` – (default: True) boolean

OUTPUT:
list of SandpileDivisors (or lists representing divisors)

EXAMPLES:

```sage
s = sandpiles.Complete(3)
sage: s.superstables(False)
[[0, 0], [0, 1], [1, 0]]
sage: s.jacobian_representatives(False)
[[0, 0, 0], [-1, 0, 1], [-1, 1, 0]]
sage: s.picard_representatives(3, False)
[[3, 0, 0], [2, 0, 1], [2, 1, 0]]
```

points()
Generators for the multiplicative group of zeros of the sandpile ideal.

OUTPUT:
list of complex numbers

EXAMPLES:
The sandpile group in this example is cyclic, and hence there is a single generator for the group of solutions.

```sage
S = sandpiles.Complete(4)
sage: S.points()
[[-I, I, 1], [-I, 1, I]]
```

postulation()
The postulation number of the toppling ideal. This is the largest weight of a superstable configuration of the graph.

OUTPUT:
nonnegative integer

EXAMPLES:

```sage
s = sandpiles.Complete(4)
sage: s.postulation()
3
```

recurrents(verbose=True)
The recurrent configurations. If `verbose` is False, the configurations are converted to lists of integers.

INPUT:
verbose – (default: True) boolean

OUTPUT:
list of recurrent configurations
EXAMPLES:

```python
sage: r = Sandpile(graphs.HouseXGraph(),0).recurrents()
sage: r[:3]
[{1: 2, 2: 3, 3: 3, 4: 1}, {1: 1, 2: 3, 3: 3, 4: 0}, {1: 1, 2: 3, 3: 3, 4: 1}]
sage: sandpiles.Complete(4).recurrents(False)
[[2, 2, 2],
 [2, 2, 1],
 [2, 1, 2],
 [1, 2, 2],
 [2, 2, 0],
 [2, 0, 2],
 [0, 2, 2],
 [2, 1, 1],
 [1, 2, 1],
 [1, 1, 2],
 [2, 1, 0],
 [2, 0, 1],
 [0, 2, 1],
 [0, 1, 2]]
sage: sandpiles.Cycle(4).recurrents(False)
[[1, 1, 1], [0, 1, 1], [1, 0, 1], [1, 1, 0]]
```

---

**reduced_laplacian()**

The reduced Laplacian matrix of the graph.

**OUTPUT:**

matrix

**EXAMPLES:**

```python
sage: S = sandpiles.Diamond()
sage: S.laplacian()
[  2 -1 -1  0]
[-1  3 -1 -1]
[-1 -1  3 -1]
[  0 -1 -1  2]
sage: S.reduced_laplacian()
[ 3 -1 -1]
[-1  3 -1]
[-1 -1  2]
```

**Note:** This is the Laplacian matrix with the row and column indexed by the sink vertex removed.

---

**reorder_vertices()**

A copy of the sandpile with vertex names permuted. After reordering, vertex $u$ comes before vertex $v$ in the list of vertices if $u$ is closer to the sink.
OUTPUT:
Sandpile

EXAMPLES:

```sage
sage: S = Sandpile({0: [1], 2: [0, 1], 1: [2]})
sage: S.dict()
{0: {1: 1}, 1: {2: 1}, 2: {0: 1, 1: 1}}
sage: T = S.reorder_vertices()
```

The vertices 1 and 2 have been swapped:

```sage
sage: T.dict()
{0: {1: 1}, 1: {0: 1, 2: 1}, 2: {0: 1}}
```

---

```sage
resolution(verbos=False)
```

A minimal free resolution of the homogeneous toppling ideal. If `verbose` is `True`, then all of the mappings are returned. Otherwise, the resolution is summarized.

INPUT:

`verbose` – (default: `False`) boolean

OUTPUT:

free resolution of the toppling ideal

EXAMPLES:

```sage
sage: S = Sandpile({0: {}, 1: {0: 1, 2: 1, 3: 4}, 2: {3: 5}, 3: {1: 1, 2: 1}}, 0)
sage: S.resolution()
# a Gorenstein sandpile graph
'\text{R}^1 \leftarrow \text{R}^5 \leftarrow \text{R}^5 \leftarrow \text{R}^1'
sage: S.resolution(True)
[ [ x1^2 - x3*x0 x3*x1 - x2*x0 x3^2 - x2*x1 x2*x3 - x0^2 x2^2 - x1*x0],
 [ x3 x2 0 x0 0] [ x2^2 - x1*x0]
 [ -x1 -x3 x2 0 -x0] [-x2*x3 + x0^2]
 [ x0 x1 0 x2 0] [-x3^2 + x2*x1]
 [ 0 0 -x1 -x3 x2] [x3*x1 - x2*x0]
 [ 0 0 x0 x1 -x3], [ x1^2 - x3*x0]
]
sage: r = S.resolution(True)
sage: r[0]*r[1]
[0 0 0 0 0]
sage: r[1]*r[2]
[0]
[0]
[0]
[0]
[0]
```

---

```sage
ring()
```
The ring containing the homogeneous toppling ideal.

**OUTPUT:**

```
ring
```

**EXAMPLES:**

```
sage: S = sandpiles.Diamond()
sage: S.ring()
Multivariate Polynomial Ring in x3, x2, x1, x0 over Rational Field
sage: S.ring().gens()
(x3, x2, x1, x0)
```

**Note:** The indeterminate $x_i$ corresponds to the $i$-th vertex as listed my the method *vertices*. The term-ordering is degrevlex with indeterminates ordered according to their distance from the sink (larger indeterminates are further from the sink).

---

**show(****kwds**)

Draw the underlying graph.

**INPUT:**

`kwds` – (optional) arguments passed to the show method for Graph or DiGraph

**EXAMPLES:**

```
sage: S = Sandpile({0:[], 1:[0,3,4], 2:[0,3,5], 3:[2,5], 4:[1,1], 5:[2,4]})
sage: S.show()
sage: S.show(graph_border=True, edge_labels=True)
```

---

**show3d(****kwds**)

Draw the underlying graph.

**INPUT:**

`kwds` – (optional) arguments passed to the show method for Graph or DiGraph

**EXAMPLES:**

```
sage: S = sandpiles.House()
sage: S.show3d()
```

---

**sink()**

The sink vertex.

**OUTPUT:**

```
sink vertex
```

**EXAMPLES:**
sage: G = sandpiles.House()
sage: G.sink()
0
sage: H = sandpiles.Grid(2,2)
sage: H.sink()
(0, 0)
sage: type(H.sink())
<... 'tuple'>

—

\texttt{smith\_form()}

The Smith normal form for the Laplacian. In detail: a list of integer matrices \( D, U, V \) such that \( ULV = D \) where \( L \) is the transpose of the Laplacian, \( D \) is diagonal, and \( U \) and \( V \) are invertible over the integers.

OUTPUT:

list of integer matrices

EXAMPLES:

sage: s = sandpiles.Complete(4)
sage: D,U,V = s.smith_form()
sage: D
[[1 0 0 0]
[0 4 0 0]
[0 0 4 0]
[0 0 0 0]]
sage: U*s.laplacian()*V == D  # laplacian symmetric => transpose not necessary
True

—

\texttt{solve()}

Approximations of the complex affine zeros of the sandpile ideal.

OUTPUT:

list of complex numbers

EXAMPLES:

sage: S = Sandpile({0: {}, 1: {2: 2}, 2: {0: 4, 1: 1}}, 0)
sage: S.solve()
[[-0.707107000000000 + 0.707107000000000*I,
  -0.707107000000000 + 0.707107000000000*I],
 [-0.707107000000000 - 0.707107000000000*I,
  -0.707107000000000 - 0.707107000000000*I],
 [-I, -I],
 [I, I],
 [0.707107000000000 + 0.707107000000000*I,
  -0.707107000000000 + 0.707107000000000*I],
 [-0.707107000000000 - 0.707107000000000*I,
  -0.707107000000000 - 0.707107000000000*I],
 [1, 1],
 [-1, -1]]

(continues on next page)
Note: The solutions form a multiplicative group isomorphic to the sandpile group. Generators for this group are given exactly by `points()`.

---

**stable_configs(smax=None)**

Generator for all stable configurations. If `smax` is provided, then the generator gives all stable configurations less than or equal to `smax`. If `smax` does not represent a stable configuration, then each component of `smax` is replaced by the corresponding component of the maximal stable configuration.

**INPUT:**

- `smax` – (optional) SandpileConfig or list representing a SandpileConfig

**OUTPUT:**

generator for all stable configurations

**EXAMPLES:**

```python
sage: s = sandpiles.Complete(3)
sage: a = s.stable_configs()
sage: next(a)
{1: 0, 2: 0}
sage: [i.values() for i in a]
[[0, 1], [1, 0], [1, 1]]
sage: b = s.stable_configs([1,0])
sage: list(b)
[{{1: 0, 2: 0}, {1: 1, 2: 0}}]
```

---

**stationary_density()**

The stationary density of the sandpile.

**OUTPUT:**

rational number

**EXAMPLES:**

```python
sage: s = sandpiles.Complete(3)
sage: s.stationary_density()
10/9
sage: s = Sandpile(digraphs.DeBruijn(2,2),'00')
sage: s.stationary_density()
9/8
```
Note: The stationary density of a sandpile is the sum \( \sum_c (\text{deg}(c) + \text{deg}(s)) \) where \( \text{deg}(s) \) is the degree of the sink and the sum is over all recurrent configurations.

REFERENCES:

- [Levine2014]

---

**superstables(\texttt{verbose=\texttt{True}})**

The superstable configurations. If \texttt{verbose} is False, the configurations are converted to lists of integers. Superstables for undirected graphs are also known as G-parking functions.

**INPUT:**

\texttt{verbose – (default: True) boolean}

**OUTPUT:**

list of \texttt{SandpileConfig}

**EXAMPLES:**

```sage
sage: sp = Sandpile(graphs.HouseXGraph(),0).superstables()
sage: sp[:3]
[[1: 0, 2: 0, 3: 0, 4: 0], [1: 1, 2: 0, 3: 0, 4: 1], [1: 1, 2: 0, 3: 0, 4: 0]]
sage: sandpiles.Complete(4).superstables(False)
[[0, 0, 0], [0, 0, 1], [0, 1, 0], [0, 2, 0], [2, 0, 0], [0, 1, 1], [1, 0, 1], [0, 1, 2], [0, 2, 1], [1, 0, 2], [1, 2, 0], [2, 0, 1], [2, 1, 0]]
sage: sandpiles.Cycle(4).superstables(False)
[[0, 0, 0], [1, 0, 0], [0, 1, 0], [0, 0, 1]]
```

---

**symmetric_recurrents(\texttt{orbits})**

The symmetric recurrent configurations.

**INPUT:**

\texttt{orbits - list of lists partitioning the vertices}

**OUTPUT:**

list of recurrent configurations
EXAMPLES:

```python
sage: S = Sandpile({0: {}, 1: {0: 1, 2: 1, 3: 1}, 2: {1: 1, 3: 1, 4: 1}, 3: {1: 1, 2: 1, 4: 1}, 4: {2: 1, 3: 1}})
sage: S.symmetric_recurrents([[1], [2, 3], [4]])
[{1: 2, 2: 2, 3: 2, 4: 1}, {1: 2, 2: 2, 3: 2, 4: 0}]
sage: S.recurrents()
[{1: 2, 2: 2, 3: 2, 4: 1}, {1: 2, 2: 2, 3: 2, 4: 0}, {1: 2, 2: 1, 3: 2, 4: 0}, {1: 2, 2: 2, 3: 0, 4: 1}, {1: 2, 2: 0, 3: 2, 4: 1}, {1: 2, 2: 2, 3: 1, 4: 0}, {1: 2, 2: 1, 3: 2, 4: 1}, {1: 2, 2: 2, 3: 1, 4: 1}]
```

**Note:** The user is responsible for ensuring that the list of orbits comes from a group of symmetries of the underlying graph.

---

tutte_polynomial()

The Tutte polynomial of the underlying graph. Only defined for undirected sandpile graphs.

**OUTPUT:**

polynomial

**EXAMPLES:**

```python
sage: s = sandpiles.Complete(4)
sage: s.tutte_polynomial()
x^3 + y^3 + 3*x^2 + 4*x*y + 3*y^2 + 2*x + 2*y
sage: s.tutte_polynomial().subs(x=1)
y^3 + 3*y^2 + 6*y + 6
sage: s.tutte_polynomial().subs(x=1).coefficients() == s.h_vector()
True
```

---

unsaturated_ideal()

The unsaturated, homogeneous toppling ideal.

**OUTPUT:**

ideal

**EXAMPLES:**

```python
sage: S = sandpiles.Diamond()
sage: S.unsaturated_ideal().gens()
[x1^3 - x3*x2*x0, x2^3 - x3*x1*x0, x3^2 - x2*x1]
```
sage: S.ideal().gens()
[x2*x1 - x0^2, x3^2 - x0^2, x1^3 - x3*x2*x0, x3*x1^2 - x2^2*x0, x2^3 - x3*x1*x0, x3*x2^2 - x1^2*x0]

—

version()
The version number of Sage Sandpiles.

OUTPUT:
string

EXAMPLES:

sage: Sandpile.version()
Sage Sandpiles Version 2.4
sage: S = sandpiles.Complete(3)
sage: S.version()
Sage Sandpiles Version 2.4

—

zero_config()
The all-zero configuration.

OUTPUT:
SandpileConfig

EXAMPLES:

sage: s = sandpiles.Diamond()
sage: s.zero_config()
{1: 0, 2: 0, 3: 0}

—

zero_div()
The all-zero divisor.

OUTPUT:
SandpileDivisor

EXAMPLES:

sage: S = sandpiles.House()
sage: S.zero_div()
{0: 0, 1: 0, 2: 0, 3: 0, 4: 0}
SandpileConfig

Summary of methods.

- + — Addition of configurations.
- & — The stabilization of the sum.
- greater-equal — True if every component of self is at least that of other.
- greater — True if every component of self is at least that of other and the two configurations are not equal.
- ~ — The stabilized configuration.
- less-equal — True if every component of self is at most that of other.
- less — True if every component of self is at most that of other and the two configurations are not equal.
- * — The recurrent element equivalent to the sum.
- ^ — Exponentiation for the *-operator.
- - — The additive inverse of the configuration.
- - — Subtraction of configurations.
- add_random — Add one grain of sand to a random vertex.
- burst_size — The burst size of the configuration with respect to the given vertex.
- deg — The degree of the configuration.
- dualize — The difference with the maximal stable configuration.
- equivalent_recurrent — The recurrent configuration equivalent to the given configuration.
- equivalent_superstable — The equivalent superstable configuration.
- fire_script — Fire the given script.
- fire_unstable — Fire all unstable vertices.
- fire_vertex — Fire the given vertex.
- help — List of SandpileConfig methods.
- is_recurrent — Is the configuration recurrent?
- is_stable — Is the configuration stable?
- is_superstable — Is the configuration superstable?
- is_symmetric — Is the configuration symmetric?
- order — The order of the equivalent recurrent element.
- sandpile — The configuration's underlying sandpile.
- show — Show the configuration.
- stabilize — The stabilized configuration.
- support — The vertices containing sand.
- unstable — The unstable vertices.
- values — The values of the configuration as a list.
Complete descriptions of SandpileConfig methods.

---

+ Addition of configurations.

**INPUT:**
- `other` – SandpileConfig

**OUTPUT:**
- sum of `self` and `other`

**EXAMPLES:**

```python
sage: S = sandpiles.Cycle(3)
sage: c = SandpileConfig(S, [1,2])
sage: d = SandpileConfig(S, [3,2])
sage: c + d
{1: 4, 2: 4}
```

---

& The stabilization of the sum.

**INPUT:**
- `other` – SandpileConfig

**OUTPUT:**
- SandpileConfig

**EXAMPLES:**

```python
sage: S = sandpiles.Cycle(4)
sage: c = SandpileConfig(S, [1,0,0])
sage: c + c # ordinary addition
{1: 2, 2: 0, 3: 0}
sage: c & c # add and stabilize
{1: 0, 2: 1, 3: 0}
sage: c*c # add and find equivalent recurrent
{1: 1, 2: 1, 3: 1}
sage: ~(c + c) == c & c
True
```

---

>= True if every component of `self` is at least that of `other`.

**INPUT:**
- `other` – SandpileConfig

**OUTPUT:**
boolean

EXAMPLES:

\begin{verbatim}
sage: S = sandpiles.Cycle(3)
sage: c = SandpileConfig(S, [1,2])
sage: d = SandpileConfig(S, [2,3])
sage: e = SandpileConfig(S, [2,0])
sage: c >= c
True
sage: d >= c
True
sage: c >= d
False
sage: e >= c
False
sage: c >= e
False
\end{verbatim}

> True if every component of self is at least that of other and the two configurations are not equal.

INPUT:
other – SandpileConfig

OUTPUT:
boolean

EXAMPLES:

\begin{verbatim}
sage: S = sandpiles.Cycle(3)
sage: c = SandpileConfig(S, [1,2])
sage: d = SandpileConfig(S, [1,3])
sage: c > c
False
sage: d > c
True
sage: c > d
False
\end{verbatim}

~ The stabilized configuration.

OUTPUT:
SandpileConfig

EXAMPLES:

\begin{verbatim}
sage: S = sandpiles.House()
sage: c = S.max_stable() + S.identity()
sage: ~c == c.stabilize()
True
\end{verbatim}
\[ \leq \]

True if every component of self is at most that of other.

**INPUT:**
other – SandpileConfig

**OUTPUT:**
boolean

**EXAMPLES:**

```
sage: S = sandpiles.Cycle(3)
sage: c = SandpileConfig(S, [1,2])
sage: d = SandpileConfig(S, [2,3])
sage: e = SandpileConfig(S, [2,0])
sage: c <= c
True
sage: c <= d
True
sage: d <= c
False
sage: c <= e
False
sage: e <= c
False
```

\[ < \]

True if every component of self is at most that of other and the two configurations are not equal.

**INPUT:**
other – SandpileConfig

**OUTPUT:**
boolean

**EXAMPLES:**

```
sage: S = sandpiles.Cycle(3)
sage: c = SandpileConfig(S, [1,2])
sage: d = SandpileConfig(S, [2,3])
sage: c < c
False
sage: c < d
True
sage: d < c
False
sage: S = Sandpile(graphs.CycleGraph(3), 0)
sage: c = SandpileConfig(S, [1,2])
sage: d = SandpileConfig(S, [2,3])
sage: c < c
False
```

(continues on next page)
If \( \text{other} \) is an configuration, the recurrent element equivalent to the sum. If \( \text{other} \) is an integer, the sum of configuration with itself \( \text{other} \) times.

INPUT:

\( \text{other} \) – SandpileConfig or Integer

OUTPUT:

SandpileConfig

EXAMPLES:

\[
\begin{align*}
\text{sage: } & S = \text{sandpiles.Cycle(4)} \\
\text{sage: } & c = \text{SandpileConfig}(S, [1,0,0]) \\
\text{sage: } & c + c \quad \# \text{ ordinary addition} \\
& \{1: 2, 2: 0, 3: 0\} \\
\text{sage: } & c & c \quad \# \text{ add and stabilize} \\
& \{1: 0, 2: 1, 3: 0\} \\
\text{sage: } & c^{*}c \quad \# \text{ add and find equivalent recurrent} \\
& \{1: 1, 2: 1, 3: 1\} \\
\text{sage: } & (c^{*}c).\text{is_recurrent()}; \\
& \text{True} \\
\text{sage: } & c^{*}(-c) == S.\text{identity()}; \\
& \text{True} \\
\text{sage: } & c \\
& \{1: 1, 2: 0, 3: 0\} \\
\text{sage: } & c^{*}3 \\
& \{1: 3, 2: 0, 3: 0\}
\end{align*}
\]

The recurrent element equivalent to the sum of the configuration with itself \( k \) times. If \( k \) is negative, do the same for the negation of the configuration. If \( k \) is zero, return the identity of the sandpile group.

INPUT:

\( k \) – SandpileConfig

OUTPUT:

SandpileConfig

EXAMPLES:

\[
\begin{align*}
\text{sage: } & S = \text{sandpiles.Cycle(4)} \\
\text{sage: } & c = \text{SandpileConfig}(S, [1,0,0])
\end{align*}
\]
```
sage: c^3
{1: 1, 2: 1, 3: 0}
sage: (c + c + c) == c^3
False
sage: (c + c + c).equivalent_recurrent() == c^3
True
sage: c^(-1)
{1: 1, 2: 1, 3: 0}
sage: c^0 == S.identity()
True

The additive inverse of the configuration.
```

**OUTPUT:**
SandpileConfig

**EXAMPLES:**
```
sage: S = sandpiles.Cycle(3)
sage: c = SandpileConfig(S, [1,2])
sage: -c
{1: -1, 2: -2}
```

**Subtraction of configurations.**

**INPUT:**
other – SandpileConfig

**OUTPUT:**
sum of self and other

**EXAMPLES:**
```
sage: S = sandpiles.Cycle(3)
sage: c = SandpileConfig(S, [1,2])
sage: d = SandpileConfig(S, [3,2])
sage: c - d
{1: -2, 2: 0}
```

**add_random(distrib=None)**

Add one grain of sand to a random vertex. Optionally, a probability distribution, distrib, may be placed on the vertices or the nonsink vertices. See NOTE for details.

**INPUT:**

**distrib** – (optional) list of nonnegative numbers summing to 1 (representing a prob. dist.)

**OUTPUT:**
SandpileConfig

EXAMLES:

```
sage: s = sandpiles.Complete(4)
sage: c = s.zero_config()
sage: c.add_random() # random
{1: 0, 2: 1, 3: 0}
sage: c
{1: 0, 2: 0, 3: 0}
sage: c.add_random([0.1,0.1,0.8]) # random
{1: 0, 2: 0, 3: 1}
sage: c.add_random([0.7,0.1,0.1,0.1]) # random
{1: 0, 2: 0, 3: 0}
```

We compute the “sizes” of the avalanches caused by adding random grains of sand to the maximal stable configuration on a grid graph. The function stabilize() returns the firing vector of the stabilization, a dictionary whose values say how many times each vertex fires in the stabilization:

```
sage: S = sandpiles.Grid(10,10)
sage: m = S.max_stable()
sage: a = []
sage: for i in range(1000):
    ....:     m = m.add_random()
    ....:     m, f = m.stabilize(True)
    ....:     a.append(sum(f.values()))
sage: p = list_plot([[log(i+1),log(a.count(i))] for i in [0..max(a)] if a.count(i)])
sage: p.axes_labels(['\text{log}(N)', '\text{log}(D(N))'])
sage: t = text("Distribution of avalanche sizes", (2,2), rgbcolor=(1,0,0))
sage: show(p+t,axes_labels=['\text{log}(N)', '\text{log}(D(N))'])
```

Note: If distrib is None, then the probability is the uniform probability on the nonsink vertices. Otherwise, there are two possibilities:

(i) the length of distrib is equal to the number of vertices, and distrib represents a probability distribution on all of the vertices. In that case, the sink may be chosen at random, in which case, the configuration is unchanged.

(ii) Otherwise, the length of distrib must be equal to the number of nonsink vertices, and distrib represents a probability distribution on the nonsink vertices.

Warning: If distrib != None, the user is responsible for assuring the sum of its entries is 1 and that its length is equal to the number of sink vertices or the number of nonsink vertices.

---

burst_size(v)

The burst size of the configuration with respect to the given vertex.

INPUT:

- v – vertex

OUTPUT:
integer

EXAMPLES:

```python
sage: s = sandpiles.Diamond()
sage: [i.burst_size(0) for i in s.recurrents()]
[1, 1, 1, 1, 1, 1, 1, 1]
sage: [i.burst_size(1) for i in s.recurrents()]
[0, 0, 1, 2, 1, 2, 0, 2]
```

**Note:** To define \( c.burst(v) \), if \( v \) is not the sink, let \( c' \) be the unique recurrent for which the stabilization of \( c' + v \) is \( c \). The burst size is then the amount of sand that goes into the sink during this stabilization. If \( v \) is the sink, the burst size is defined to be 1.

REFERENCES:

- [Levine2014]

---

**deg()**

The degree of the configuration.

OUTPUT:

integer

EXAMPLES:

```python
sage: S = sandpiles.Complete(3)
sage: c = SandpileConfig(S, [1,2])
sage: c.deg()
3
```

---

**dualize()**

The difference with the maximal stable configuration.

OUTPUT:

SandpileConfig

EXAMPLES:

```python
sage: S = sandpiles.Cycle(3)
sage: c = SandpileConfig(S, [1,2])
sage: S.max_stable()
{1: 1, 2: 1}
sage: c.dualize()
{1: 0, 2: -1}
sage: S.max_stable() - c == c.dualize()
True
```

---

**equivalent_recurrent(with_firing_vector=False)**

The recurrent configuration equivalent to the given configuration. Optionally, return the corresponding firing vector.
Thematic Tutorials, Release 9.7

**equivalent_recurrent(with_firing_vector=False)**

The equivalent recurrent configuration. Optionally, return the corresponding firing vector.

**INPUT:**

with_firing_vector – (default: False) boolean

**OUTPUT:**

SandpileConfig or [SandpileConfig, firing_vector]

**EXAMPLES:**

```
sage: S = sandpiles.Diamond()
sage: c = SandpileConfig(S, [0,0,0])
sage: c.equivalent_recurrent() == S.identity()
True
sage: x = c.equivalent_recurrent(True)
sage: r = vector([x[0][v] for v in S.nonsink_vertices()])
sage: f = vector([x[1][v] for v in S.nonsink_vertices()])
sage: cv = vector(c.values())
sage: r == cv - f*S.reduced_laplacian()
True
```

**Note:** Let $L$ be the reduced Laplacian, $c$ the initial configuration, $r$ the returned configuration, and $f$ the firing vector. Then $r = c - f \cdot L$.

**equivalent_superstable(with_firing_vector=False)**

The equivalent superstable configuration. Optionally, return the corresponding firing vector.

**INPUT:**

with_firing_vector – (default: False) boolean

**OUTPUT:**

SandpileConfig or [SandpileConfig, firing_vector]

**EXAMPLES:**

```
sage: S = sandpiles.Diamond()
sage: m = S.max_stable()
sage: m.equivalent_superstable().is_superstable()
True
sage: x = m.equivalent_superstable(True)
sage: s = vector(x[0].values())
sage: f = vector(x[1].values())
sage: mv = vector(m.values())
sage: s == mv - f*S.reduced_laplacian()
True
```

**Note:** Let $L$ be the reduced Laplacian, $c$ the initial configuration, $s$ the returned configuration, and $f$ the firing vector. Then $s = c - f \cdot L$.

**fire_script(sigma)**
Fire the given script. In other words, fire each vertex the number of times indicated by \( \sigma \).

**INPUT:**

\( \sigma \) – SandpileConfig or (list or dict representing a SandpileConfig)

**OUTPUT:**

SandpileConfig

**EXAMPLES:**

```python
sage: S = sandpiles.Cycle(4)
sage: c = SandpileConfig(S, [1,2,3])
sage: c.unstable()
[2, 3]
sage: c.fire_script(SandpileConfig(S,[0,1,1]))
{1: 2, 2: 1, 3: 2}
sage: c.fire_script(SandpileConfig(S,[2,0,0])) == c.fire_vertex(1).fire_vertex(1)
True
```

__fire_unstable()__

Fire all unstable vertices.

**OUTPUT:**

SandpileConfig

**EXAMPLES:**

```python
sage: S = sandpiles.Cycle(4)
sage: c = SandpileConfig(S, [1,2,3])
sage: c.fire_unstable()
{1: 2, 2: 1, 3: 2}
```

__fire_vertex(v)__

Fire the given vertex.

**INPUT:**

\( v \) – vertex

**OUTPUT:**

SandpileConfig

**EXAMPLES:**

```python
sage: S = sandpiles.Cycle(3)
sage: c = SandpileConfig(S, [1,2])
sage: c.fire_vertex(2)
{1: 2, 2: 0}
```

__help(\texttt{verbose=True})__

List of SandpileConfig methods. If verbose, include short descriptions.
**INPUT:**

verbose – (default: True) boolean

**OUTPUT:**

printed string

**EXAMPLES:**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Sage: SandpileConfig.help()</code></td>
<td>Shortcuts for SandpileConfig operations:</td>
</tr>
<tr>
<td>~c</td>
<td>-- stabilize</td>
</tr>
<tr>
<td>c &amp; d</td>
<td>-- add and stabilize</td>
</tr>
<tr>
<td>c * c</td>
<td>-- add and find equivalent recurrent</td>
</tr>
<tr>
<td>c^k</td>
<td>-- add k times and find equivalent recurrent (taking inverse if k is negative)</td>
</tr>
</tbody>
</table>

For detailed help with any method FOO listed below, enter "SandpileConfig.FOO?" or enter "c.FOO?" for any SandpileConfig c.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add_random</td>
<td>-- Add one grain of sand to a random vertex.</td>
</tr>
<tr>
<td>burst_size</td>
<td>-- The burst size of the configuration with respect to the given vertex.</td>
</tr>
<tr>
<td>deg</td>
<td>-- The degree of the configuration.</td>
</tr>
<tr>
<td>dualize</td>
<td>-- The difference with the maximal stable configuration.</td>
</tr>
<tr>
<td>equivalent_recurrent</td>
<td>-- The recurrent configuration equivalent to the given configuration.</td>
</tr>
<tr>
<td>equivalent_superstable</td>
<td>-- The equivalent superstable configuration.</td>
</tr>
<tr>
<td>fire_script</td>
<td>-- Fire the given script.</td>
</tr>
<tr>
<td>fire_unstable</td>
<td>-- Fire all unstable vertices.</td>
</tr>
<tr>
<td>fire_vertex</td>
<td>-- Fire the given vertex.</td>
</tr>
<tr>
<td>help</td>
<td>-- List of SandpileConfig methods.</td>
</tr>
<tr>
<td>is_recurrent</td>
<td>-- Is the configuration recurrent?</td>
</tr>
<tr>
<td>is_stable</td>
<td>-- Is the configuration stable?</td>
</tr>
<tr>
<td>is_superstable</td>
<td>-- Is the configuration superstable?</td>
</tr>
<tr>
<td>is_symmetric</td>
<td>-- Is the configuration symmetric?</td>
</tr>
<tr>
<td>order</td>
<td>-- The order of the equivalent recurrent element.</td>
</tr>
<tr>
<td>sandpile</td>
<td>-- The configuration's underlying sandpile.</td>
</tr>
<tr>
<td>show</td>
<td>-- Show the configuration.</td>
</tr>
<tr>
<td>stabilize</td>
<td>-- The stabilized configuration.</td>
</tr>
<tr>
<td>support</td>
<td>-- The vertices containing sand.</td>
</tr>
<tr>
<td>unstable</td>
<td>-- The unstable vertices.</td>
</tr>
<tr>
<td>values</td>
<td>-- The values of the configuration as a list.</td>
</tr>
</tbody>
</table>

**is_recurrent()**

Is the configuration recurrent?

**OUTPUT:**

boolean

**EXAMPLES:**
sage: S = sandpiles.Diamond()
sage: S.identity().is_recurrent()
True
sage: S.zero_config().is_recurrent()
False

—

is_stable()
Is the configuration stable?

OUTPUT:
boolean

EXAMPLES:

sage: S = sandpiles.Diamond()
sage: S.max_stable().is_stable()
True
sage: (2*S.max_stable()).is_stable()
False
sage: (S.max_stable() & S.max_stable()).is_stable()
True

—

is_superstable()
Is the configuration superstable?

OUTPUT:
boolean

EXAMPLES:

sage: S = sandpiles.Diamond()
sage: S.zero_config().is_superstable()
True

—

is_symmetric(orbits)
Is the configuration symmetric? Return True if the values of the configuration are constant over the vertices in each sublist of orbits.

INPUT:

orbits – list of lists of vertices

OUTPUT:
boolean

EXAMPLES:

sage: S = Sandpile({0: {},
....: 1: {0: 1, 2: 1, 3: 1},
....: 2: {1: 1, 3: 1, 4: 1},
(continues on next page)
Thematic Tutorials, Release 9.7

(continued from previous page)

....:
3: {1: 1, 2: 1, 4: 1},
....:
4: {2: 1, 3: 1}}

```
sage: c = SandpileConfig(S, [1, 2, 2, 3])
sage: c.is_symmetric([[2,3]])
True
```

—

**order()**

The order of the equivalent recurrent element.

**OUTPUT:**

integer

**EXAMPLES:**

```
sage: S = sandpiles.Diamond()
sage: c = SandpileConfig(S,[2,0,1])
sage: c.order()
4
sage: ~(c + c + c + c) == S.identity()
True
sage: c = SandpileConfig(S,[1,1,0])
sage: c.order()
1
sage: c.is_recurrent()
False
sage: c.equivalent_recurrent() == S.identity()
True
```

—

**sandpile()**

The configuration’s underlying sandpile.

**OUTPUT:**

Sandpile

**EXAMPLES:**

```
sage: S = sandpiles.Diamond()
sage: c = S.identity()
sage: c.sandpile()
Diamond sandpile graph: 4 vertices, sink = 0
sage: c.sandpile() == S
True
```

—

**show(sink=True, colors=True, heights=False, directed=None, **kwds)**

Show the configuration.

**INPUT:**

- sink – (default: True) whether to show the sink
• **colors** – (default: True) whether to color-code the amount of sand on each vertex

• **heights** – (default: False) whether to label each vertex with the amount of sand

• **directed** – (optional) whether to draw directed edges

• **kwds** – (optional) arguments passed to the show method for Graph

**EXAMPLES:**

```python
sage: S = sandpiles.Diamond()
sage: c = S.identity()
sage: c.show()
sage: c.show(directed=False)
sage: c.show(sink=False, colors=False, heights=True)
```

---

**stabilize(with_firing_vector=False)**

The stabilized configuration. Optionally returns the corresponding firing vector.

**INPUT:**

with_firing_vector – (default: False) boolean

**OUTPUT:**

SandpileConfig or [SandpileConfig, firing_vector]

**EXAMPLES:**

```python
sage: S = sandpiles.House()
sage: c = 2*S.max_stable()
sage: c._set_stabilize()
sage: '_stabilize' in c.__dict__
True
sage: S = sandpiles.House()
sage: c = S.max_stable() + S.identity()
sage: c.stabilize(True)

[(1: 1, 2: 2, 3: 2, 4: 1), {1: 2, 2: 2, 3: 3, 4: 3}]
sage: S.max_stable() & S.identity() == c.stabilize()
True
sage: ~c == c.stabilize()
True
```

---

**support()**

The vertices containing sand.

**OUTPUT:**

list - support of the configuration

**EXAMPLES:**

```python
sage: S = sandpiles.Diamond()
sage: c = S.identity()
sage: c
{1: 2, 2: 2, 3: 0}
```
Thematic Tutorials, Release 9.7

sage: c.support()
[1, 2]

---

unstable()
The unstable vertices.

OUTPUT:
list of vertices

EXAMPLES:

sage: S = sandpiles.Cycle(4)
sage: c = SandpileConfig(S, [1,2,3])
sage: c.unstable()
[2, 3]

---

values()
The values of the configuration as a list. The list is sorted in the order of the vertices.

OUTPUT:
list of integers
boolean

EXAMPLES:

sage: S = Sandpile({'a':[c,b], 'b':[c,a], 'c':[a],'a'})
sage: c = SandpileConfig(S, {'b':1, 'c':2})
sage: c
{'b': 1, 'c': 2}
sage: c.values()
[1, 2]
sage: S.nonsink_vertices()
['b', 'c']

---

SandpileDivisor

Summary of methods.

- + — Addition of divisors.
  - greater-equal — True if every component of self is at least that of other.
  - greater — True if every component of self is at least that of other and the two divisors are not equal.
  - less-equal — True if every component of self is at most that of other.
  - less — True if every component of self is at most that of other and the two divisors are not equal.
  - - — The additive inverse of the divisor.
• - — Subtraction of divisors.
• Dcomplex — The support-complex.
• add_random — Add one grain of sand to a random vertex.
• betti — The Betti numbers for the support-complex.
• deg — The degree of the divisor.
• dualize — The difference with the maximal stable divisor.
• effective_div — All linearly equivalent effective divisors.
• fire_script — Fire the given script.
• fire_unstable — Fire all unstable vertices.
• fire_vertex — Fire the given vertex.
• help — List of SandpileDivisor methods.
• is_alive — Is the divisor stabilizable?
• is_linearly_equivalent — Is the given divisor linearly equivalent?
• is_q_reduced — Is the divisor q-reduced?
• is_symmetric — Is the divisor symmetric?
• is_weierstrass_pt — Is the given vertex a Weierstrass point?
• polytope — The polytope determining the complete linear system.
• polytope_integer_pts — The integer points inside divisor’s polytope.
• q_reduced — The linearly equivalent q-reduced divisor.
• rank — The rank of the divisor.
• sandpile — The divisor’s underlying sandpile.
• show — Show the divisor.
• simulate_threshold — The first unstabilizable divisor in the closed Markov chain.
• stabilize — The stabilization of the divisor.
• support — List of vertices at which the divisor is nonzero.
• unstable — The unstable vertices.
• values — The values of the divisor as a list.
• weierstrass_div — The Weierstrass divisor.
• weierstrass_gap_seq — The Weierstrass gap sequence at the given vertex.
• weierstrass_pts — The Weierstrass points (vertices).
• weierstrass_rank_seq — The Weierstrass rank sequence at the given vertex.

Complete descriptions of SandpileDivisor methods.

—

+
Addition of divisors.

INPUT:
other – SandpileDivisor

OUTPUT:
sum of self and other

EXAMPLES:

```python
sage: S = sandpiles.Cycle(3)
sage: D = SandpileDivisor(S, [1,2,3])
sage: E = SandpileDivisor(S, [3,2,1])
sage: D + E
{0: 4, 1: 4, 2: 4}
```

>=

True if every component of self is at least that of other.

INPUT:
other – SandpileDivisor

OUTPUT:
boolean

EXAMPLES:

```python
sage: S = sandpiles.Cycle(3)
sage: D = SandpileDivisor(S, [1,2,3])
sage: E = SandpileDivisor(S, [2,3,4])
sage: F = SandpileDivisor(S, [2,0,4])
sage: D >= D
True
sage: E >= D
True
sage: D >= E
False
sage: F >= D
False
sage: D >= F
False
```

>

True if every component of self is at least that of other and the two divisors are not equal.

INPUT:
other – SandpileDivisor

OUTPUT:
boolean

EXAMPLES:
\begin{verbatim}
sage: S = sandpiles.Cycle(3)
sage: D = SandpileDivisor(S, [1,2,3])
sage: E = SandpileDivisor(S, [1,3,4])
sage: D > D
False
sage: E > D
True
sage: D > E
False
\end{verbatim}

\textbf{<=}

True if every component of \texttt{self} is at most that of \texttt{other}.

\textbf{INPUT:}

\texttt{other} – SandpileDivisor

\textbf{OUTPUT:}

boolean

\textbf{EXAMPLES:}

\begin{verbatim}
sage: S = sandpiles.Cycle(3)
sage: D = SandpileDivisor(S, [1,2,3])
sage: E = SandpileDivisor(S, [2,3,4])
sage: F = SandpileDivisor(S, [2,0,4])
sage: D <= D
True
sage: D <= E
True
sage: E <= D
False
sage: D <= F
False
sage: F <= D
False
\end{verbatim}

\textbf{<}

True if every component of \texttt{self} is at most that of \texttt{other} and the two divisors are not equal.

\textbf{INPUT:}

\texttt{other} – SandpileDivisor

\textbf{OUTPUT:}

boolean

\textbf{EXAMPLES:}

\begin{verbatim}
sage: S = sandpiles.Cycle(3)
sage: D = SandpileDivisor(S, [1,2,3])
sage: E = SandpileDivisor(S, [2,3,4])
\end{verbatim}
The additive inverse of the divisor.

**OUTPUT:**
SandpileDivisor

**EXAMPLES:**

```python
sage: S = sandpiles.Cycle(3)
sage: D = SandpileDivisor(S, [1,2,3])
sage: -D
{0: -1, 1: -2, 2: -3}
```

Subtraction of divisors.

**INPUT:**
other – SandpileDivisor

**OUTPUT:**
Difference of self and other

**EXAMPLES:**

```python
sage: S = sandpiles.Cycle(3)
sage: D = SandpileDivisor(S, [1,2,3])
sage: E = SandpileDivisor(S, [3,2,1])
sage: D - E
{0: -2, 1: 0, 2: 2}
```

**Dcomplex()**

The support-complex. (See NOTE.)

**OUTPUT:**
simplicial complex

**EXAMPLES:**

```python
sage: S = sandpiles.House()
sage: p = SandpileDivisor(S, [1,2,1,0,0]).Dcomplex()
sage: p.homology()
```
\{(0: 0, 1: Z × Z, 2: 0)\}
\texttt{sage: p.f_vector()}
\[1, 5, 10, 4\]
\texttt{sage: p.betti()}
\{0: 1, 1: 2, 2: 0\}

\textbf{Note:} The “support-complex” is the simplicial complex determined by the supports of the linearly equivalent effective divisors.

---

\textbf{add_random(distrib=None)}

Add one grain of sand to a random vertex.

\textbf{INPUT:}

\texttt{distrib} – (optional) list of nonnegative numbers representing a probability distribution on the vertices

\textbf{OUTPUT:}

SandpileDivisor

\textbf{EXAMPLES:}

\begin{verbatim}
\texttt{sage: s = sandpiles.Complete(4)}
\texttt{sage: D = s.zero_div()}
\texttt{sage: D.add_random()}  # random
\{0: 0, 1: 0, 2: 1, 3: 0\}
\texttt{sage: D.add_random([0.1, 0.1, 0.1, 0.7])}  # random
\{0: 0, 1: 0, 2: 0, 3: 1\}
\end{verbatim}

\textbf{Warning:} If \texttt{distrib} is not \texttt{None}, the user is responsible for assuring the sum of its entries is 1.

---

\textbf{betti()}

The Betti numbers for the support-complex. (See NOTE.)

\textbf{OUTPUT:}

dictionary of integers

\textbf{EXAMPLES:}

\begin{verbatim}
\texttt{sage: S = sandpiles.Cycle(3)}
\texttt{sage: D = SandpileDivisor(S, [2, 0, 1])}
\texttt{sage: D.betti()}
\{0: 1, 1: 1\}
\end{verbatim}

\textbf{Note:} The “support-complex” is the simplicial complex determined by the supports of the linearly equivalent effective divisors.
---

**deg()**

The degree of the divisor.

**OUTPUT:**

integer

**EXAMPLES:**

```
sage: S = sandpiles.Cycle(3)
sage: D = SandpileDivisor(S, [1,2,3])
sage: D.deg()
6
```

---

**dualize()**

The difference with the maximal stable divisor.

**OUTPUT:**

SandpileDivisor

**EXAMPLES:**

```
sage: S = sandpiles.Cycle(3)
sage: D = SandpileDivisor(S, [1,2,3])
sage: D.dualize()
{0: 0, 1: -1, 2: -2}
sage: S.max_stable_div() - D == D.dualize()
True
```

---

**effective_div(***verbose*=True, **with_firing_vectors**=False)**

All linearly equivalent effective divisors. If *verbose* is False, the divisors are converted to lists of integers. If *with_firing_vectors* is True then a list of firing vectors is also given, each of which prescribes the vertices to be fired in order to obtain an effective divisor.

**INPUT:**

- **verbose** – (default: True) boolean
- **with_firing_vectors** – (default: False) boolean

**OUTPUT:**

list (of divisors)

**EXAMPLES:**

```
sage: s = sandpiles.Complete(4)
sage: D = SandpileDivisor(s,[4,2,0,0])
sage: sorted(D.effective_div(), key=str)
[{
0: 0, 1: 2, 2: 0, 3: 4},
{
0: 0, 1: 6, 2: 0, 3: 0},
{
0: 1, 1: 3, 2: 1, 3: 1},
,...
```
Thematic Tutorials, Release 9.7

```
{0: 2, 1: 0, 2: 2, 3: 2},
{0: 4, 1: 2, 2: 0, 3: 0}]
sage: sorted(D.effective_div(False))
[[0, 2, 0, 4],
 [0, 2, 4, 0],
 [0, 6, 0, 0],
 [1, 3, 1, 1],
 [2, 0, 2, 2],
 [4, 2, 0, 0]]
sage: sorted(D.effective_div(with_firing_vectors=True), key=str)
[({0: 0, 1: 2, 2: 0, 3: 4}, (0, -1, -1, -2)),
 ({0: 0, 1: 2, 2: 4, 3: 0}, (0, -1, -2, -1)),
 ({0: 0, 1: 6, 2: 0, 3: 0}, (0, -2, -1, -1)),
 ({0: 1, 1: 3, 2: 1, 3: 1}, (0, -1, -1, -1)),
 ({0: 2, 1: 0, 2: 3, 3: 2}, (0, 0, -1, -1)),
 ({0: 4, 1: 2, 2: 0, 3: 0}, (0, 0, 0, 0))]
sage: a = _[2]
sage: a[0].values()
[0, 6, 0, 0]
sage: vector(D.values()) - s.laplacian()*a[1]
(0, 6, 0, 0)
sage: sorted(D.effective_div(False, True))
[[[0, 2, 0, 4], (0, -1, -1, -2)),
 [[0, 2, 4, 0], (0, -1, -2, -1)),
 [[0, 6, 0, 0], (0, -2, -1, -1)),
 [[1, 3, 1, 1], (0, -1, -1, -1)),
 [[2, 0, 2, 2], (0, 0, -1, -1)),
 [[4, 2, 0, 0], (0, 0, 0, 0)]]
sage: D = SandpileDivisor(s,[-1,0,0,0])
sage: D.effective_div(False, True)
[]
```

**fire_script(sigma)**

Fire the given script. In other words, fire each vertex the number of times indicated by sigma.

**INPUT:**

sigma – SandpileDivisor or (list or dict representing a SandpileDivisor)

**OUTPUT:**

SandpileDivisor

**EXAMPLES:**

```
sage: S = sandpiles.Cycle(3)
sage: D = SandpileDivisor(S, [1,2,3])
sage: D.unstable()
[1, 2]
sage: D.fire_script([0,1,1])
{0: 3, 1: 1, 2: 2}
sage: D.fire_script(SandpileDivisor(S,[2,0,0])) == D.fire_vertex(0).fire_vertex(0)
True
```
—

**fire_unstable()**

Fire all unstable vertices.

**OUTPUT:**

SandpileDivisor

**EXAMPLES:**

```
sage: S = sandpiles.Cycle(3)
sage: D = SandpileDivisor(S, [1,2,3])
sage: D.fire_unstable()
{0: 3, 1: 1, 2: 2}
```

—

**fire_vertex(v)**

Fire the given vertex.

**INPUT:**

v – vertex

**OUTPUT:**

SandpileDivisor

**EXAMPLES:**

```
sage: S = sandpiles.Cycle(3)
sage: D = SandpileDivisor(S, [1,2,3])
sage: D.fire_vertex(1)
{0: 2, 1: 0, 2: 4}
```

—

**help(verbose=True)**

List of SandpileDivisor methods. If **verbose**, include short descriptions.

**INPUT:**

verbose – (default: True) boolean

**OUTPUT:**

printed string

**EXAMPLES:**

```
sage: SandpileDivisor.help()
```

For detailed help with any method FOO listed below, enter "SandpileDivisor.FOO?" or enter "D.FOO?" for any SandpileDivisor D.

```
Dcomplex       -- The support-complex.
add_random    -- Add one grain of sand to a random vertex.
betti         -- The Betti numbers for the support-complex.
deg           -- The degree of the divisor.
dualize       -- The difference with the maximal stable divisor.
```

(continues on next page)
Effective_div -- All linearly equivalent effective divisors.
fire_script -- Fire the given script.
fire_unstable -- Fire all unstable vertices.
fire_vertex -- Fire the given vertex.
help -- List of SandpileDivisor methods.
is_alive -- Is the divisor stabilizable?
is_linearly_equivalent -- Is the given divisor linearly equivalent?
is_q_reduced -- Is the divisor q-reduced?
is_symmetric -- Is the divisor symmetric?
is_weierstrass_pt -- Is the given vertex a Weierstrass point?
polytope -- The polytope determining the complete linear system.
polytope_integer_pts -- The integer points inside divisor's polytope.
q_reduced -- The linearly equivalent q-reduced divisor.
rank -- The rank of the divisor.
sandpile -- The divisor's underlying sandpile.
show -- Show the divisor.
simulate_threshold -- The first unstabilizable divisor in the closed Markov chain.
stabilize -- The stabilization of the divisor.
support -- List of vertices at which the divisor is nonzero.
unstable -- The unstable vertices.
values -- The values of the divisor as a list.
weierstrass_div -- The Weierstrass divisor.
weierstrass_gap_seq -- The Weierstrass gap sequence at the given vertex.
weierstrass_pts -- The Weierstrass points (vertices).
weierstrass_rank_seq -- The Weierstrass rank sequence at the given vertex.

---

**is_alive(cycle=False)**

Is the divisor stabilizable? In other words, will the divisor stabilize under repeated firings of all unstable vertices? Optionally returns the resulting cycle.

**INPUT:**

cycle – (default: False) boolean

**OUTPUT:**

boolean or optionally, a list of SandpileDivisors

**EXAMPLES:**

```
sage: S = sandpiles.Complete(4)
sage: D = SandpileDivisor(S, {0: 4, 1: 3, 2: 3, 3: 2})
sage: D.is_alive()
True
sage: D.is_alive(True)
[[{0: 4, 1: 3, 2: 3, 3: 2}, {0: 3, 1: 2, 2: 2, 3: 5}, {0: 1, 1: 4, 2: 4, 3: 3}]

---

**is_linearly_equivalent(D, with_firing_vector=False)**

Is the given divisor linearly equivalent? Optionally, returns the firing vector. (See NOTE.)

**INPUT:**

• \(D\) – SandpileDivisor or list, tuple, etc. representing a divisor

• with_firing_vector – (default: False) boolean

OUTPUT:
boolean or integer vector

EXAMPLES:

```python
sage: s = sandpiles.Complete(3)
sage: D = SandpileDivisor(s,[2,0,0])
sage: D.is_linearly_equivalent([0,1,1])
True
sage: D.is_linearly_equivalent([0,1,1],True)
(0, -1, -1)
sage: v = vector(D.is_linearly_equivalent([0,1,1],True))
sage: vector(D.values()) - s.laplacian()*v
(0, 1, 1)
sage: D.is_linearly_equivalent([0,0,0])
False
sage: D.is_linearly_equivalent([0,0,0],True)
()  
```

Note:
• If with_firing_vector is False, returns either True or False.

• If with_firing_vector is True then: (i) if self is linearly equivalent to \(D\), returns a vector \(v\) such that self - v*self.laplacian().transpose() = D. Otherwise, (ii) if self is not linearly equivalent to \(D\), the output is the empty vector, ().

---

is_q_reduced()

Is the divisor \(q\)-reduced? This would mean that \(self = c + kq\) where \(c\) is superstable, \(k\) is an integer, and \(q\) is the sink vertex.

OUTPUT:
boolean

EXAMPLES:

```python
sage: s = sandpiles.Complete(4)
sage: D = SandpileDivisor(s,[2,-3,2,0])
sage: D.is_q_reduced()
False
sage: SandpileDivisor(s,[10,0,1,2]).is_q_reduced()
True
```

For undirected or, more generally, Eulerian graphs, \(q\)-reduced divisors are linearly equivalent if and only if they are equal. The same does not hold for general directed graphs:

```python
sage: s = Sandpile({0:[1],1:[1,1]})
sage: D = SandpileDivisor(s,[-1,1])
sage: Z = s.zero_div()
```
sage: D.is_q_reduced()
True
sage: Z.is_q_reduced()
True
sage: D == Z
False
sage: D.is_linearly_equivalent(Z)
True

—

is_symmetric(orbits)

Is the divisor symmetric? Return True if the values of the configuration are constant over the vertices in each sublist of orbits.

INPUT:

orbits – list of lists of vertices

OUTPUT:

boolean

EXAMPLES:

sage: S = sandpiles.House()
sage: S.dict()
{0: {1: 1, 2: 1},
  1: {0: 1, 3: 1},
  2: {0: 1, 3: 1, 4: 1},
  3: {1: 1, 2: 1, 4: 1},
  4: {2: 1, 3: 1}}
sage: D = SandpileDivisor(S, [0,0,1,1,3])
sage: D.is_symmetric([[2,3], [4]])
True

—

is_weierstrass_pt(v='sink')

Is the given vertex a Weierstrass point?

INPUT:

v – (default: sink) vertex

OUTPUT:

boolean

boolean

EXAMPLES:

sage: s = sandpiles.House()
sage: K = s.canonical_divisor()
sage: K.weierstrass_rank_seq()  # sequence at the sink vertex, 0
(1, 0, -1)
sage: K.is_weierstrass_pt()
False
\textbf{K} \texttt{.weierstrass_rank_seq(4)}
\begin{verbatim}
 sage: K.weierstrass_rank_seq(4)
 (1, 0, 0, -1)
\end{verbatim}
\textbf{K} \texttt{.is_weierstrass_pt(4)}
\begin{verbatim}
 sage: K.is_weierstrass_pt(4)
 True
\end{verbatim}

\textbf{Note:} The vertex $v$ is a (generalized) Weierstrass point for divisor $D$ if the sequence of ranks $r(D - nv)$ for $n = 0, 1, 2, \ldots$ is not $r(D), r(D) - 1, \ldots, 0, -1, -1, \ldots$.

---

\textbf{polytope()}

The polytope determining the complete linear system.

\textbf{OUTPUT:}

polytope

\textbf{EXAMPLES:}

\begin{verbatim}
sage: s = sandpiles.Complete(4)
sage: D = SandpileDivisor(s,[4,2,0,0])
sage: p = D.polytope()
sage: p.inequalities()
(An inequality (-3, 1, 1) x + 2 >= 0,
 An inequality (1, 1, 1) x + 4 >= 0,
 An inequality (1, -3, 1) x + 0 >= 0,
 An inequality (1, 1, -3) x + 0 >= 0)
sage: D = SandpileDivisor(s,[-1,0,0,0])
sage: D.polytope()
The empty polyhedron in QQ^3
\end{verbatim}

\textbf{Note:} For a divisor $D$, this is the intersection of (i) the polyhedron determined by the system of inequalities $L^tx \leq D$ where $L^t$ is the transpose of the Laplacian with (ii) the hyperplane $x_{\text{sink,vertex}} = 0$. The polytope is thought of as sitting in $(n-1)$-dimensional Euclidean space where $n$ is the number of vertices.

---

\textbf{polytope_integer_pts()}

The integer points inside divisor's polytope. The polytope referred to here is the one determining the divisor's complete linear system (see the documentation for \texttt{polytope}).

\textbf{OUTPUT:}

tuple of integer vectors

\textbf{EXAMPLES:}

\begin{verbatim}
sage: s = sandpiles.Complete(4)
sage: D = SandpileDivisor(s,[4,2,0,0])
sage: sorted(D.polytope_integer_pts())
[(-2, -1, -1),
 (-1, -2, -1),
\end{verbatim}
\[ (-1, -1, -2), \]
\[ (-1, -1, -1), \]
\[ (0, -1, -1), \]
\[ (0, 0, 0) \]
\[
\text{sage: } D = \text{SandpileDivisor}(s, [-1, 0, 0, 0])
\]
\[
\text{sage: } D.\text{polytope_integer_pts}()
\]

\[ -1, -2, 1, 2, 0 \]

\section*{q\textunderscore reduced(\texttt{verbose=True})}

The linearly equivalent \( q \)-reduced divisor.

\textbf{INPUT:}

\texttt{verbose} – (default: \texttt{True}) boolean

\textbf{OUTPUT:}

\texttt{SandpileDivisor or list representing SandpileDivisor}

\textbf{EXAMPLES:}

\[
\text{sage: } s = \text{sandpiles.Complete(4)}
\]
\[
\text{sage: } D = \text{SandpileDivisor}(s, [2, -3, 2, 0])
\]
\[
\text{sage: } D.\text{q\textunderscore reduced}()
\]
\[
\{0: -2, 1: 1, 2: 2, 3: 0\}
\]
\[
\text{sage: } D.\text{q\textunderscore reduced(False)}
\]
\[
[-2, 1, 2, 0]
\]

\textbf{Note: } The divisor \( D \) is \texttt{q\textunderscore reduced\texttt{if}} where \( c \) is superstable, \( k \) is an integer, and \( q \) is the sink.

\section*{rank(\texttt{with\_witness=False})}

The rank of the divisor. Optionally returns an effective divisor \( E \) such that \( D - E \) is not winnable (has an empty complete linear system).

\textbf{INPUT:}

\texttt{with\_witness} – (default: \texttt{False}) boolean

\textbf{OUTPUT:}

integer or (integer, \texttt{SandpileDivisor})

\textbf{EXAMPLES:}

\[
\text{sage: } S = \text{sandpiles.Complete(4)}
\]
\[
\text{sage: } D = \text{SandpileDivisor}(S, [4, 2, 0, 0])
\]
\[
\text{sage: } D.\text{rank}()
\]
\[
3
\]
\[
\text{sage: } D.\text{rank(True)}
\]
\[
(3, \{0: 3, 1: 0, 2: 1, 3: 0\})
\]
\[
\text{sage: } E = _[1]
\]
\[
\text{sage: } (D - E).\text{rank}()
\]
Riemann-Roch theorem:

sage: D.rank() - (S.canonical_divisor()-D).rank() == D.deg() + 1 - S.genus()
True

Riemann-Roch theorem:

sage: D.rank() - (S.canonical_divisor()-D).rank() == D.deg() + 1 - S.genus()
True
sage: S = Sandpile({0: [1, 1, 2], 1: [0, 0, 0, 1, 1, 1, 2, 2], 2: [2, 2, 1, 1, 0]}, 0) # multigraph
˓→with loops
sage: D = SandpileDivisor(S, [4, 2, 0])
sage: D.rank(True)
(2, {0: 1, 1: 1, 2: 1})
sage: S = Sandpile({0: [1, 2], 1: [0, 2, 2], 2: [0, 1]}, 0) # directed graph
sage: S.is_undirected()
False
sage: D = SandpileDivisor(S, [0, 2, 0])
sage: D.effective_div()

Note: The rank of a divisor $D$ is -1 if $D$ is not linearly equivalent to an effective divisor (i.e., the dollar game represented by $D$ is unwinnable). Otherwise, the rank of $D$ is the largest integer $r$ such that $D - E$ is linearly equivalent to an effective divisor for all effective divisors $E$ with $\deg(E) = r$.

sandpile()
The divisor’s underlying sandpile.

OUTPUT:
Sandpile

EXAMPLES:

```
sage: S = sandpiles.Diamond()
sage: D = SandpileDivisor(S, [1, -2, 0, 3])
sage: D.sandpile()
Diamond sandpile graph: 4 vertices, sink = 0
sage: D.sandpile() == S
True
```

show(heights=True, directed=None, **kwds)
Show the divisor.

INPUT:

- **heights** – (default: True) whether to label each vertex with the amount of sand
- **directed** – (optional) whether to draw directed edges
- **kwds** – (optional) arguments passed to the show method for Graph

EXAMPLES:

```python
sage: S = sandpiles.Diamond()
sage: D = SandpileDivisor(S, [1, -2, 0, 2])
sage: D.show(graph_border=True, vertex_size=700, directed=False)
```

---

**simulate_threshold**(distrib=None)

The first unstabilizable divisor in the closed Markov chain. (See NOTE.)

INPUT:

- **distrib** – (optional) list of nonnegative numbers representing a probability distribution on the vertices

OUTPUT:

SandpileDivisor

EXAMPLES:

```python
sage: s = sandpiles.Complete(4)
sage: D = s.zero_div()
sage: D.simulate_threshold()  # random
{0: 2, 1: 3, 2: 1, 3: 2}
sage: n(mean([D.simulate_threshold().deg() for _ in range(10)]))  # random
7.10000000000000
sage: n(s.stationary_density()*s.num_verts())
6.93750000000000
```

**Note:** Starting at **self**, repeatedly choose a vertex and add a grain of sand to it. Return the first unstabilizable divisor that is reached. Also see the markov_chain method for the underlying sandpile.

---

**stabilize**(with_firing_vector=False)

The stabilization of the divisor. If not stabilizable, return an error.

INPUT:

- **with_firing_vector** – (default: False) boolean

EXAMPLES:

```python
sage: s = sandpiles.Complete(4)
sage: D = SandpileDivisor(s, [0, 3, 0, 0])
sage: D.stabilize()
{0: 1, 1: 0, 2: 1, 3: 1}
```
Thematic Tutorials, Release 9.7

sage: D.stabilize(with_firing_vector=True)

```
[{
    0: 1, 1: 0, 2: 1, 3: 1},
   {
    0: 0, 1: 1, 2: 0, 3: 0}
]
```

__support__
List of vertices at which the divisor is nonzero.

OUTPUT:
list representing the support of the divisor

EXAMPLES:

```
sage: S = sandpiles.Cycle(4)
sage: D = SandpileDivisor(S, [0,0,1,1])
sage: D.support()
[2, 3]
sage: S.vertices(sort=True)
[0, 1, 2, 3]
```

__unstable__
The unstable vertices.

OUTPUT:
list of vertices

EXAMPLES:

```
sage: S = sandpiles.Cycle(3)
sage: D = SandpileDivisor(S, [1,2,3])
sage: D.unstable()
[1, 2]
```

__values__
The values of the divisor as a list. The list is sorted in the order of the vertices.

OUTPUT:
list of integers

boolean

EXAMPLES:

```
sage: S = Sandpile({'a': ['c', 'b'], 'b': ['c', 'a'], 'c': ['a']}, 'a')
sage: D = SandpileDivisor(S, {'a': 0, 'b': 1, 'c': 2})
sage: D.values()
[0, 1, 2]
sage: S.vertices(sort=True)
['a', 'b', 'c']
```
weierstrass_div(\texttt{verbose}=True)

The Weierstrass divisor. Its value at a vertex is the weight of that vertex as a Weierstrass point. (See \texttt{SandpileDivisor.weierstrass_gap_seq}.)

**INPUT:**

\texttt{verbose} – (default: True) boolean

**OUTPUT:**

\texttt{SandpileDivisor}

**EXAMPLES:**

```sage
s = sandpiles.Diamond()
sage: D = SandpileDivisor(s,[4,2,1,0])
sage: [D.weierstrass_rank_seq(v) for v in s]
[(5, 4, 3, 2, 1, 0, 0, -1),
 (5, 4, 3, 2, 1, 0, 0, -1),
 (5, 4, 3, 2, 1, 0, 0, -1),
 (5, 4, 3, 2, 1, 0, 0, -1)]
sage: D.weierstrass_div()
{0: 1, 1: 0, 2: 2, 3: 1}
sage: k5 = sandpiles.Complete(5)
sage: K = k5.canonical_divisor()
sage: K.weierstrass_div()
{0: 9, 1: 9, 2: 9, 3: 9, 4: 9}
```

weierstrass_gap_seq(\texttt{v}=\textquote{sink}, \texttt{weight}=True)

The Weierstrass gap sequence at the given vertex. If \texttt{weight} is True, then also compute the weight of each gap value.

**INPUT:**

\begin{itemize}
  \item \texttt{v} – (default: sink) vertex
  \item \texttt{weight} – (default: True) boolean
\end{itemize}

**OUTPUT:**

list or (list of list) of integers

**EXAMPLES:**

```sage
s = sandpiles.Cycle(4)
sage: D = SandpileDivisor(s,[2,0,0,0])
sage: [D.weierstrass_gap_seq(v,False) for v in s.vertices(sort=True)]
[(1, 3), (1, 2), (1, 3), (1, 2)]
sage: [D.weierstrass_gap_seq(v) for v in s.vertices(sort=True)]
[((1, 3), 1), ((1, 2), 0), ((1, 3), 1), ((1, 2), 0)]
sage: D.weierstrass_gap_seq()  # gap sequence at sink vertex, 0
((1, 3), 1)
sage: D.weierstrass_rank_seq()  # rank sequence at the sink vertex
(1, 0, 0, -1)
```
Note: The integer $k$ is a Weierstrass gap for the divisor $D$ at vertex $v$ if the rank of $D - (k - 1)v$ does not equal the rank of $D - kv$. Let $r$ be the rank of $D$ and let $k_i$ be the $i$-th gap at $v$. The Weierstrass weight of $v$ for $D$ is the sum of $(k_i - i)$ as $i$ ranges from 1 to $r + 1$. It measures the difference between the sequence $r, r - 1, ..., 0, -1, -1, ...$ and the rank sequence $\text{rank}(D), \text{rank}(D - v), \text{rank}(D - 2v), ...$.

\[\text{weierstrass_pts(with_rank_seq=False)}\]

The Weierstrass points (vertices). Optionally, return the corresponding rank sequences.

**INPUT:**

with_rank_seq – (default: False) boolean

**OUTPUT:**

tuple of vertices or list of (vertex, rank sequence)

**EXAMPLES:**

```
sage: s = sandpiles.House()
sage: K = s.canonical_divisor()
sage: K.weierstrass_pts()
(4,)
sage: K.weierstrass_pts(True)
[(4, (1, 0, 0, -1))]
```

Note: The vertex $v$ is a (generalized) Weierstrass point for divisor $D$ if the sequence of ranks $r(D - nv)$ for $n = 0, 1, 2, ...$ is not $r(D), r(D) - 1, ..., 0, -1, -1, ...$.

\[\text{weierstrass_rank_seq(v='sink')}\]

The Weierstrass rank sequence at the given vertex. Computes the rank of the divisor $D - nv$ starting with $n = 0$ and ending when the rank is $-1$.

**INPUT:**

v – (default: sink) vertex

**OUTPUT:**

tuple of int

**EXAMPLES:**

```
sage: s = sandpiles.House()
sage: K = s.canonical_divisor()
sage: [K.weierstrass_rank_seq(v) for v in s.vertices(sort=True)]
[(1, 0, -1), (1, 0, -1), (1, 0, -1), (1, 0, -1), (1, 0, 0, -1)]
```

---
Other

• **firing_graph** — The firing graph.
• **parallel_firing_graph** — The parallel-firing graph.
• **sandpiles** — Some examples of sandpiles.
• **wilmes_algorithm** — Find matrix with the same integer row span as $M$ that is the reduced Laplacian of a digraph.

Complete descriptions of methods.

**firing_graph**(S, eff)

Creates a digraph with divisors as vertices and edges between two divisors $D$ and $E$ if firing a single vertex in $D$ gives $E$.

**INPUT:**

S – Sandpile eff – list of divisors

**OUTPUT:**

DiGraph

**EXAMPLES:**

```python
sage: S = sandpiles.Cycle(6)
sage: D = SandpileDivisor(S, [1,1,1,1,2,0])
sage: eff = D.effective_div()
sage: firing_graph(S,eff).show3d(edge_size=.005,vertex_size=0.01)
```

---

**parallel_firing_graph**(S, eff)

Creates a digraph with divisors as vertices and edges between two divisors $D$ and $E$ if firing all unstable vertices in $D$ gives $E$.

**INPUT:**

S - Sandpile eff - list of divisors

**OUTPUT:**

DiGraph

**EXAMPLES:**

```python
sage: S = Sandpile(graphs.CycleGraph(6),0)
sage: D = SandpileDivisor(S, [1,1,1,1,2,0])
sage: eff = D.effective_div()
sage: parallel_firing_graph(S,eff).show3d(edge_size=.005,vertex_size=0.01)
```

---

**sandpiles**

Some examples of sandpiles.

Here are the available examples; you can also type “sandpiles.” and hit tab to get a list:

• “Complete()”
• “Cycle()”
• “Diamond()”
• “Grid()”
• “House()”

EXAMPLES:
```
sage: s = sandpiles.Complete(4)
sage: s.invariant_factors()
[1, 4, 4]
sage: s.laplacian()
[ 3 -1 -1 -1]
[-1 3 -1 -1]
[-1 -1 3 -1]
[-1 -1 -1 3]
```

---

wilmes_algorithm(M)

Computes an integer matrix \( L \) with the same integer row span as \( M \) and such that \( L \) is the reduced laplacian of a directed multigraph.

INPUT:

\( M \) - square integer matrix of full rank

OUTPUT:

\( L \) - integer matrix

EXAMPLES:
```
sage: P = matrix([[2,3,-7,-3],[5,2,-5,5],[8,2,5,4],[-5,-9,6,6]])
sage: wilmes_algorithm(P)
[ 3279 -79 -1599 -1600]
[ -1 1539 -136 -1402]
[ 0 -1 1650 -1649]
[ 0 0 -1658 1658]
```

NOTE:

The algorithm is due to John Wilmes.

Help

Documentation for each method is available through the Sage online help system:
```
sage: SandpileConfig.fire_vertex?
Base Class: <class 'instancemethod'>
String Form: <unbound method SandpileConfig.fire_vertex>
Namespace: Interactive
File: /usr/local/sage-4.7/local/lib/python2.6/site-packages/sage/sandpiles/__init__.py
Definition: SandpileConfig.fire_vertex(self, v)
Docstring: (continues on next page)
```
Fire the vertex `v`.

**INPUT:**

`v` - vertex

**OUTPUT:**

SandpileConfig

**EXAMPLES:**

```python
sage: S = Sandpile(graphs.CycleGraph(3), 0)
sage: c = SandpileConfig(S, [1,2])
sage: c.fire_vertex(2)
{1: 2, 2: 0}
```

**Note:** An alternative to `SandpileConfig.fire_vertex`? in the preceding code example would be `c.`

```python
fire_vertex
```

if `c` is any SandpileConfig.

Enter `Sandpile.help`, `SandpileConfig.help`, and `SandpileDivisor.help()` for lists of available

Sandpile-specific methods.


**Contact**

Please contact davidp@reed.edu with questions, bug reports, and suggestions for additional features and other improvements.

### 12.1.3 Group Theory and Sage

**Author: Robert A. Beezer, University of Puget Sound**

This compilation collects Sage commands that are useful for a student in an introductory course on group theory. It is not intended to teach Sage or to teach group theory. (There are many introductory texts on group theory and more information on Sage can be found via www.sagemath.org) Rather, by presenting commands roughly in the order a student would learn the corresponding mathematics they might be encouraged to experiment and learn more about mathematics and learn more about Sage. Not coincidentally, when Sage was the acronym SAGE, the “E” in Sage stood for “Experimentation.”

This guide is also distributed in PDF format and as a Sage worksheet. The worksheet version can be imported into the Sage notebook environment running in a web browser, and then the displayed chunks of code may be executed by Sage if one clicks on the small “evaluate” link below each cell, for a fully interactive experience. A PDF and Sage worksheet versions of this tutorial are available at http://abstract.ups.edu/sage-aata.html.

**Table of contents**

- Group Theory and Sage
Basic properties of the integers

Integer division

The command \( a \% b \) will return the remainder upon division of \( a \) by \( b \). In other words, the value is the unique integer \( r \) such that:

1. \( 0 \leq r < b \); and
2. \( a = bq + r \) for some integer \( q \) (the quotient).

Then \( (a - r)/b \) will equal \( q \). For example:

```
sage: r = 14 % 3
sage: q = (14 - r) / 3
sage: r, q
(2, 4)
```

will return 2 for the value of \( r \) and 4 for the value of \( q \). Note that the “/” is integer division, where any remainder is cast away and the result is always an integer. So, for example, \( 14 / 3 \) will again equal 4, not 4.66666.

Greatest common divisor

The greatest common divisor of \( a \) and \( b \) is obtained with the command \( \text{gcd}(a, b) \), where in our first uses, \( a \) and \( b \) are integers. Later, \( a \) and \( b \) can be other objects with a notion of divisibility and “greatness,” such as polynomials. For example:

```
sage: gcd(2776, 2452)
4
```
Extended greatest common divisor

The command \texttt{xgcd(a, b)} ("eXtended GCD") returns a triple where the first element is the greatest common divisor of \(a\) and \(b\) (as with the \texttt{gcd(a, b)} command above), but the next two elements are the values of \(r\) and \(s\) such that \(ra + sb = \text{gcd}(a, b)\). For example, \texttt{xgcd(633, 331)} returns \((1, 194, -371)\). Portions of the triple can be extracted using \([\ ]\) to access the entries of the triple, starting with the first as number 0. For example, the following should return the result True (even if you change the values of \(a\) and \(b\)). Studying this block of code will go a long way towards helping you get the most out of Sage’s output. (Note that “=” is how a value is assigned to a variable, while as in the last line, “==” is how we determine equality of two items.)

\begin{verbatim}
sage: a = 633
dsage: b = 331
dsage: extended = xgcd(a, b)
sage: g = extended[0]
sage: r = extended[1]
sage: s = extended[2]
sage: g == r*a + s*b
True
\end{verbatim}

Divisibility

A remainder of zero indicates divisibility. So \((a \ % \ b) \ == \ 0\) will return True if \(b\) divides \(a\), and will otherwise return False. For example, \((9 \ % \ 3) \ == \ 0\) is True, but \((9 \ % \ 4) \ == \ 0\) is False. Try predicting the output of the following before executing it in Sage.

\begin{verbatim}
sage: answer1 = ((20 \ % \ 5) == 0)
sage: answer2 = ((17 \ % \ 4) == 0)
sage: answer1, answer2
(True, False)
\end{verbatim}

Factoring

As promised by the Fundamental Theorem of Arithmetic, \texttt{factor(a)} will return a unique expression for \(a\) as a product of powers of primes. It will print in a nicely-readable form, but can also be manipulated with Python as a list of pairs \((p_i, e_i)\) containing primes as bases, and their associated exponents. For example:

\begin{verbatim}
sage: factor(2600)
2^3 * 5^2 * 13
\end{verbatim}

If you just want the prime divisors of an integer, then use the \texttt{prime_divisors(a)} command, which will return a list of all the prime divisors of \(a\). For example:

\begin{verbatim}
sage: prime_divisors(2600)
[2, 5, 13]
\end{verbatim}

We can strip off other pieces of the prime decomposition using two levels of \([\ ]\). This is another good example to study in order to learn about how to drill down into Python lists.

\begin{verbatim}
sage: n = 2600
sage: decomposition = factor(n)
sage: print("{} decomposes as {}".format(n, decomposition))
\end{verbatim}
2600 decomposes as $2^3 \times 5^2 \times 13$

```python
sage: secondterm = decomposition[1]
sage: print("Base and exponent (pair) for second prime: "+str(secondterm))
Base and exponent (pair) for second prime: (5, 2)
sage: base = secondterm[0]
sage: exponent = secondterm[1]
sage: print("Base is "+str(base))
Base is 5
sage: print("Exponent is "+str(exponent))
Exponent is 2
sage: thirdbase = decomposition[2][0]
sage: thirdexponent = decomposition[2][1]
sage: print("Base of third term is {} with exponent {}".format(thirdbase, thirdexponent))
Base of third term is 13 with exponent 1
```

With a bit more work, the `factor()` command can be used to factor more complicated items, such as polynomials.

**Multiplicative inverse, modular arithmetic**

The command `inverse_mod(a, n)` yields the multiplicative inverse of $a$ mod $n$ (or an error if it doesn’t exist). For example:

```python
sage: inverse_mod(352, 917)
508
```

(As a check, find the integer $m$ such that $352 \times 508 \equiv m \times 917 + 1$. Then try

```python
sage: inverse_mod(4, 24)
Traceback (most recent call last):
  ... ZeroDivisionError: inverse of Mod(4, 24) does not exist
```

and explain the result.

**Powers with modular arithmetic**

The command `power_mod(a, m, n)` yields $a^m$ mod $n$. For example:

```python
sage: power_mod(15, 831, 23)
10
```

If $m = -1$, then this command will duplicate the function of `inverse_mod()`.
Euler $\phi$-function

The command `euler_phi(n)` will return the number of positive integers less than $n$ and relatively prime to $n$ (i.e. having greatest common divisor with $n$ equal to 1). For example:

```sage
sage: euler_phi(345)
176
```

Experiment by running the following code several times:

```sage
sage: m = random_prime(10000)
sage: n = random_prime(10000)
sage: euler_phi(m*n) == euler_phi(m) * euler_phi(n)
True
```

Feel a conjecture coming on? Can you generalize this result?

Primes

The command `is_prime(a)` returns `True` or `False` depending on if $a$ is prime or not. For example,

```sage
sage: is_prime(117371)
True
```

while

```sage
sage: is_prime(14547073)
False
```

since $14547073 = 1597 \times 9109$ (as you could determine with the `factor()` command).

The command `random_prime(a, True)` will return a random prime between 2 and $a$. Experiment with:

```sage
sage: p = random_prime(10^21, True)
sage: is_prime(p)
True
```

(Replacing `True` by `False` will speed up the search, but there will be a very small probability the result will not be prime.)

The command `prime_range(a, b)` returns an ordered list of all the primes from $a$ to $b - 1$, inclusive. For example,

```sage
sage: prime_range(500, 550)
[503, 509, 521, 523, 541, 547]
```

The commands `next_prime(a)` and `previous_prime(a)` are other ways to get a single prime number of a desired size. Give them a try.
Permutation groups

A good portion of Sage's support for group theory is based on routines from GAP (Groups, Algorithms, and Programming at https://www.gap-system.org). Groups can be described in many different ways, such as sets of matrices or sets of symbols subject to a few defining relations. A very concrete way to represent groups is via permutations (one-to-one and onto functions of the integers 1 through $n$), using function composition as the operation in the group. Sage has many routines designed to work with groups of this type and they are also a good way for those learning group theory to gain experience with the basic ideas of group theory. For both these reasons, we will concentrate on these types of groups.

Writing permutations

Sage uses “disjoint cycle notation” for permutations, see any introductory text on group theory (such as Judson, Section 4.1) for more on this. Composition occurs left to right, which is not what you might expect and is exactly the reverse of what Judson and many others use. (There are good reasons to support either direction, you just need to be certain you know which one is in play.) There are two ways to write the permutation $\sigma = (1\ 3)\ (2\ 5\ 4)$:

1. As a text string (include quotes): "(1,3) (2,5,4)"
2. As a Python list of “tuples”: [(1,3), (2,5,4)]

Groups

Sage knows many popular groups as sets of permutations. More are listed below, but for starters, the full “symmetric group” of all possible permutations of 1 through $n$ can be built with the command SymmetricGroup(n).

Permutation elements Elements of a group can be created, and composed, as follows

```
sage: G = SymmetricGroup(5)
sage: sigma = G("(1,3) (2,5,4)"")
sage: rho = G([(2,4), (1,5)])
sage: rho^(-1) * sigma * rho
(1,2,4)(3,5)
```

Available functions for elements of a permutation group include finding the order of an element, i.e. for a permutation $\sigma$ the order is the smallest power of $k$ such that $\sigma^k$ equals the identity element (). For example:

```
sage: G = SymmetricGroup(5)
sage: sigma = G("(1,3) (2,5,4)"")
sage: sigma.order()
6
```

The sign of the permutation $\sigma$ is defined to be 1 for an even permutation and $-1$ for an odd permutation. For example:

```
sage: G = SymmetricGroup(5)
sage: sigma = G("(1,3) (2,5,4)"")
sage: sigma.sign()
-1
```

since $\sigma$ is an odd permutation.

Many more available functions that can be applied to a permutation can be found via “tab-completion.” With sigma defined as an element of a permutation group, in a Sage cell, type sigma. (Note the “.”) and then press the tab key. You will get a list of available functions (you may need to scroll down to see the whole list). Experiment and explore! It is what Sage is all about. You really cannot break anything.
Creating groups

This is an annotated list of some small well-known permutation groups that can be created simply in Sage. You can find more in the source code file

```
SAGE_ROOT/src/sage/groups/perm_gps/permgroup_named.py
```

- **SymmetricGroup(n)**: All \(n!\) permutations on \(n\) symbols.
- **DihedralGroup(n)**: Symmetries of an \(n\)-gon. Rotations and flips, \(2n\) in total.
- **CyclicPermutationGroup(n)**: Rotations of an \(n\)-gon (no flips), \(n\) in total.
- **AlternatingGroup(n)**: Alternating group on \(n\) symbols having \(n!/2\) elements.
- **KleinFourGroup()**: The non-cyclic group of order 4.

Group functions

Individual elements of permutation groups are important, but we primarily wish to study groups as objects on their own. So a wide variety of computations are available for groups. Define a group, for example

```
sage: H = DihedralGroup(6)
sage: H
Dihedral group of order 12 as a permutation group
```

and then a variety of functions become available.

After trying the examples below, experiment with tab-completion. Having defined \(H\), type \(H.\) (note the “.”) and then press the tab key. You will get a list of available functions (you may need to scroll down to see the whole list). As before, *experiment and explore*—it is really hard to break anything.

Here is another couple of ways to experiment and explore. Find a function that looks interesting, say \(is\_abelian()\). Type \(H.is\_abelian?\) (note the question mark) followed by the enter key. This will display a portion of the source code for the \(is\_abelian()\) function, describing the inputs and output, possibly illustrated with example uses.

If you want to learn more about how Sage works, or possibly extend its functionality, then you can start by examining the complete Python source code. For example, try \(H.is\_abelian??\), which will allow you to determine that the \(is\_abelian()\) function is basically riding on GAP’s \(IsAbelian()\) command and asking GAP do the heavy-lifting for us. (To get the maximum advantage of using Sage it helps to know some basic Python programming, but it is not required.)

OK, on to some popular command for groups. If you are using the worksheet, be sure you have defined the group \(H\) as the dihedral group \(D_6\), since we will not keep repeating its definition below.

**Abelian?**

The command

```
sage: H = DihedralGroup(6)
sage: H.is_abelian()
False
```

will return \(False\) since \(D_6\) is a non-abelian group.
Order

The command

```python
sage: H = DihedralGroup(6)
sage: H.order()
12
```

will return 12 since $D_6$ is a group of with 12 elements.

All elements

The command

```python
sage: H = DihedralGroup(6)
sage: H.list()
[(),
 (1,5,3)(2,6,4),
 (1,3,5)(2,4,6),
 (1,6,5,4,3,2),
 (1,4)(2,5)(3,6),
 (1,2,3,4,5,6),
 (2,6)(3,5),
 (1,5)(2,4),
 (1,3)(4,6),
 (1,6)(2,5)(3,4),
 (1,4)(2,3)(5,6),
 (1,2)(3,6)(4,5)]
```

will return all of the elements of $H$ in a fixed order as a Python list. Indexing (`[]`) can be used to extract the individual elements of the list, remembering that counting the elements of the list begins at zero.

```python
sage: H = DihedralGroup(6)
sage: elements = H.list()
sage: elements[2]
(1,3,5)(2,4,6)
```

Cayley table

The command

```python
sage: H = DihedralGroup(6)
sage: H.cayley_table()
* | a b c d e f g h i j k l
---+------------------------
a | a b c d e f g h i j k l
b | b a d c f e h g j i l k
c | c k a e d g f i h l b j
d | d l b f c h e j g k a i
e | e j k g a i d l f b c h
f | f i l h b j c k e a d g
```
(continues on next page)
will construct the Cayley table (or “multiplication table”) of $H$. By default the table uses lowercase Latin letters to name the elements of the group. The actual elements used can be found using the row_keys() or column_keys() commands for the table. For example to determine the fifth element in the table, the element named e:

```
sage: H = DihedralGroup(6)
sage: T = H.cayley_table()
sage: headings = T.row_keys()
sage: headings[4]
(1,3)(4,6)
```

**Center**

The command $H$.center() will return a subgroup that is the center of the group $H$ (see Exercise 2.46 in Judson). Try

```
sage: H = DihedralGroup(6)
sage: H.center().list()
[(), (1,4)(2,5)(3,6)]
```

to see which elements of $H$ commute with *every* element of $H$.

**Cayley graph**

For fun, try `show(H.cayley_graph())`.

**Subgroups**

**Cyclic subgroups**

If $G$ is a group and $a$ is an element of the group (try $a = G$.random_element()), then

```
a = G.random_element()
H = G.subgroup([a])
```

will create $H$ as the cyclic subgroup of $G$ with generator $a$.

For example the code below will:

1. create $G$ as the symmetric group on five symbols;
2. specify $\sigma$ as an element of $G$;
3. use $\sigma$ as the generator of a cyclic subgroup $H$;
4. list all the elements of $H$.

In more mathematical notation, we might write $\langle (1\ 2\ 3)(4\ 5) \rangle = H \subseteq G = S_5$. 

12.1. Thematic tutorial document tree
sage: G = SymmetricGroup(5)
sage: sigma = G("(1,2,3) (4,5)")
sage: H = G.subgroup([sigma])
sage: H.list()
[(), (1,2,3)(4,5), (1,3,2), (4,5), (1,2,3), (1,3,2)(4,5)]

Experiment by trying different permutations for \(\sigma\) and observing the effect on \(H\).

Cyclic groups

Groups that are cyclic themselves are both important and rich in structure. The command \texttt{CyclicPermutationGroup(n)} will create a permutation group that is cyclic with \(n\) elements. Consider the following example (note that the indentation of the third line is critical) which will list the elements of a cyclic group of order 20, preceded by the order of each element.

\begin{verbatim}
sage: n = 20
sage: CN = CyclicPermutationGroup(n)
sage: for g in CN:
    ....:     print("{} {}").format(g.order(), g)
1 ()
20 (1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20)
10 (1,3,5,7,9,11,13,15,17,19)(2,4,6,8,10,12,14,16,18,20)
20 (1,4,7,10,13,16,19,2,5,8,11,14,17,20,3,6,9,12,15,18)
10 (1,5,9,13,17)(2,6,10,14,18)(3,7,11,15,19)(4,8,12,16,20)
5 (1,6,11,16)(2,7,12,17)(3,8,13,18)(4,9,14,19)(5,10,15,20)
20 (1,7,13,19,5,11,17,3,9,15)(2,8,14,20,6,12,18,4,10,16)
10 (1,8,15,2,9,16,3,10,17,4,11,18,5,12,19,6,13,20,7,14)
5 (1,9,17,5,13)(2,10,18,6,14)(3,11,19,7,15)(4,12,20,8,16)
20 (1,10,19,8,17,6,15,4,13,2,11,9,18,7,16,5,14,3,12)
20 (1,12,3,14,5,16,7,18,9,20,11,2,13,4,15,6,17,8,19,10)
10 (1,13,5,17,9)(2,14,6,18,10)(3,15,7,19,11)(4,16,8,20,12)
5 (1,14,7,20,13,6,19,12,5,18,11,4,17,10,3,16,9,2,15,8)
20 (1,15,9,3,17,11,5,19,13,7)(2,16,10,4,18,12,6,20,14,8)
4 (1,16,11,6)(2,17,12,7)(3,18,13,8)(4,19,14,9)(5,20,15,10)
5 (1,17,13,9,5)(2,18,14,10,6)(3,19,15,11,7)(4,20,16,12,8)
20 (1,18,15,12,9,6,3,20,17,14,11,8,5,2,19,16,13,10,8,7,4)
10 (1,19,17,15,13,11,9,7,5,3)(2,20,18,16,14,12,10,8,6,4)
20 (1,20,19,18,17,16,15,14,13,12,11,10,9,8,7,6,5,4,3,2)
\end{verbatim}

By varying the size of the group (change the value of \(n\)) you can begin to illustrate some of the structure of a cyclic group (for example, try a prime).

We can cut/paste an element of order 5 from the output above (in the case when the cyclic group has 20 elements) and quickly build a subgroup:

\begin{verbatim}
sage: C20 = CyclicPermutationGroup(20)
sage: rho = C20("(1,17,13,9,5)(2,18,14,10,6)(3,19,15,11,7)(4,20,16,12,8)")
sage: H = C20.subgroup([rho])
sage: H.list()
[(), (1,17,13,9,5)(2,18,14,10,6)(3,19,15,11,7)(4,20,16,12,8), (continues on next page)]
\end{verbatim}
For a cyclic group, the following command will list all of the subgroups.

```sage
sage: C20 = CyclicPermutationGroup(20)
sage: C20.conjugacy_classes_subgroups()
[Subgroup generated by [()] of (Cyclic group of order 20 as a permutation group),
 Subgroup generated by [(1,11)(2,12)(3,13)(4,14)(5,15)(6,16)(7,17)(8,18)(9,19)(10,20)] of (Cyclic group of order 20 as a permutation group),
 Subgroup generated by [(1,6,11,16)(2,7,12,17)(3,8,13,18)(4,9,14,19)(5,10,15,20), (1,11)(2,12)(3,13)(4,14)(5,15)(6,16)(7,17)(8,18)(9,19)(10,20)] of (Cyclic group of order 20 as a permutation group),
 Subgroup generated by [(1,5,9,13,17)(2,6,10,14,18)(3,7,11,15,19)(4,8,12,16,20)] of (Cyclic group of order 20 as a permutation group),
 Subgroup generated by [(1,3,5,7,9,11,13,15,17,19)(2,4,6,8,10,12,14,16,18,20), (1,5,9,13,17)(2,6,10,14,18)(3,7,11,15,19)(4,8,12,16,20)] of (Cyclic group of order 20 as a permutation group),
 Subgroup generated by [(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20), (1,3,5,7,9,11,13,15,17,19)(2,4,6,8,10,12,14,16,18,20), (1,5,9,13,17)(2,6,10,14,18)(3,7,11,15,19)(4,8,12,16,20)] of (Cyclic group of order 20 as a permutation group)]
```

Be careful, this command uses some more advanced ideas and will not usually list all of the subgroups of a group. Here we are relying on special properties of cyclic groups (but see the next section).

If you are viewing this as a PDF, you can safely skip over the next bit of code. However, if you are viewing this as a worksheet in Sage, then this is a place where you can experiment with the structure of the subgroups of a cyclic group. In the input box, enter the order of a cyclic group (numbers between 1 and 40 are good initial choices) and Sage will list each subgroup as a cyclic group with its generator. The factorization at the bottom might help you formulate a conjecture.

```python
%auto
@interact
def _(n = input_box(default=12, label = "Cyclic group of order:", type=Integer) ):
    cyclic = CyclicPermutationGroup(n)
    subgroups = cyclic.conjugacy_classes_subgroups()
    html("All subgroups of a cyclic group of order $%s$\n" % latex(n))
    table = "$\begin{array}{ll}"
    for sg in subgroups:
        table = table + latex(sg.order()) + " & \left\langle" + latex(sg.gens()[0]) + "\right\rangle\\"
    table = table + "\end{array}$"
    html(table)
    html("Hint: $%s$ factors as $%s$" % ( latex(n), latex(factor(n)) ))
```
All subgroups

If $H$ is a subgroup of $G$ and $g \in G$, then $gHg^{-1} = \{ghg^{-1} | h \in G\}$ will also be a subgroup of $G$. If $G$ is a group, then the command `G.conjugacy_classes_subgroups()` will return a list of subgroups of $G$, but not all of the subgroups. However, every subgroup can be constructed from one on the list by the $gHg^{-1}$ construction with a suitable $g$. As an illustration, the code below:

1. creates $K$ as the dihedral group of order 24, $D_{12}$;
2. stores the list of subgroups output by `K.conjugacy_classes_subgroups()` in the variable `sg`;
3. prints the elements of the list;
4. selects the second subgroup in the list, and lists its elements.

```sage
K = DihedralGroup(12)
sage: sg = K.conjugacy_classes_subgroups()
sage: sg
Subgroup generated by [()] of (Dihedral group of order 24 as a permutation group),
Subgroup generated by [(1,7)(2,8)(3,9)(4,10)(5,11)(6,12)] of (Dihedral group of order 24 as a permutation group),
Subgroup generated by [(2,12)(3,11)(4,10)(5,9)(6,8)] of (Dihedral group of order 24 as a permutation group),
Subgroup generated by [(1,2)(3,12)(4,11)(5,10)(6,9)(7,8)] of (Dihedral group of order 24 as a permutation group),
Subgroup generated by [(1,3,5,7,9,11)(2,4,6,8,10,12), (1,5,9)(2,6,10)(3,7,11)(4,8,12)] of (Dihedral group of order 24 as a permutation group),
Subgroup generated by [(2,12)(3,11)(4,10)(5,9,6,8), (1,7)(2,8)(3,9)(4,10)(5,11)(6,12)] of (Dihedral group of order 24 as a permutation group),
Subgroup generated by [(1,7)(2,8)(3,9)(4,10)(5,11)(6,12), (1,10,7,4)(2,11,8,5)(3,12,9,6)] of (Dihedral group of order 24 as a permutation group),
Subgroup generated by [(2,12)(3,11)(4,10)(5,9)(6,8), (1,5,9)(2,6,10)(3,7,11)(4,8,12)] of (Dihedral group of order 24 as a permutation group),
Subgroup generated by [(1,2)(3,12)(4,11)(5,10)(6,9)(7,8), (1,5,9)(2,6,10)(3,7,11)(4,8,12)] of (Dihedral group of order 24 as a permutation group),
Subgroup generated by [(2,12)(3,11)(4,10)(5,9)(6,8), (1,7)(2,8)(3,9)(4,10)(5,11)(6,12)] of (Dihedral group of order 24 as a permutation group),
Subgroup generated by [(1,3,5,7,9,11)(2,4,6,8,10,12), (1,5,9)(2,6,10)(3,7,11)(4,8,12)] of (Dihedral group of order 24 as a permutation group),
Subgroup generated by [(2,12)(3,11)(4,10)(5,9,6,8), (1,7)(2,8)(3,9)(4,10)(5,11)(6,12)] of (Dihedral group of order 24 as a permutation group),
Subgroup generated by [(1,2)(3,12)(4,11)(5,10)(6,9)(7,8), (1,5,9)(2,6,10)(3,7,11)(4,8,12)] of (Dihedral group of order 24 as a permutation group),
Subgroup generated by [(2,12)(3,11)(4,10)(5,9)(6,8), (1,3,5,7,9,11)(2,4,6,8,10,12), (1,5,9)(2,6,10)(3,7,11)(4,8,12)] of (Dihedral group of order 24 as a permutation group),
Subgroup generated by [(1,2)(3,12)(4,11)(5,10)(6,9)(7,8), (1,3,5,7,9,11)(2,4,6,8,10,12), (1,5,9)(2,6,10)(3,7,11)(4,8,12)] of (Dihedral group of order 24 as a permutation group),
```

```sage
sage: print("An order two subgroup:
\n\n{}").format(sg[1].list())
An order two subgroup:
[(), (1,7)(2,8)(3,9)(4,10)(5,11)(6,12)]
```
It is important to note that this is a nice long list of subgroups, but will rarely create every such subgroup. For example, the code below:

1. creates \( \rho \) as an element of the group \( K \);
2. creates \( L \) as a cyclic subgroup of \( K \);
3. prints the two elements of \( L \); and finally
4. tests to see if this subgroup is part of the output of the list \( \text{sg} \) created just above (it is not).

```
sage: K = DihedralGroup(12)
sage: sg = K.conjugacy_classes_subgroups()
sage: rho = K("(1,4) (2,3) (5,12) (6,11) (7,10) (8,9)")
sage: L = PermutationGroup([rho])
sage: L.list()
[() , (1,4)(2,3)(5,12)(6,11)(7,10)(8,9)]
sage: L in sg
False
```

Symmetry groups

You can give Sage a short list of elements of a permutation group and Sage will find the smallest subgroup that contains those elements. We say the list “generates” the subgroup. We list a few interesting subgroups you can create this way.

**Symmetries of an equilateral triangle**

Label the vertices of an equilateral triangle as 1, 2 and 3. Then any permutation of the vertices will be a symmetry of the triangle. So either SymmetricGroup(3) or DihedralGroup(3) will create the full symmetry group.

**Symmetries of an \( n \)-gon**

A regular, \( n \)-sided figure in the plane (an \( n \)-gon) has \( 2n \) symmetries, comprised of \( n \) rotations (including the trivial one) and \( n \) “flips” about various axes. The dihedral group \( \text{DihedralGroup}(n) \) is frequently defined as exactly the symmetry group of an \( n \)-gon.

**Symmetries of a tetrahedron**

Label the 4 vertices of a regular tetrahedron as 1, 2, 3 and 4. Fix the vertex labeled 4 and rotate the opposite face through 120 degrees. This will create the permutation/symmetry \( (1\ 2\ 3) \). Similarly, fixing vertex 1, and rotating the opposite face will create the permutation \( (2\ 3\ 4) \). These two permutations are enough to generate the full group of the twelve symmetries of the tetrahedron. Another symmetry can be visualized by running an axis through the midpoint of an edge of the tetrahedron through to the midpoint of the opposite edge, and then rotating by 180 degrees about this axis. For example, the 1–2 edge is opposite the 3–4 edge, and the symmetry is described by the permutation \( (1\ 2)(3\ 4) \). This permutation, along with either of the above permutations will also generate the group. So here are two ways to create this group:

```
sage: tetra_one = PermutationGroup(["(1,2,3)"", "(2,3,4)"\])
sage: tetra_one
Permutation Group with generators [(2,3,4), (1,2,3)]
sage: tetra_two = PermutationGroup(["(1,2,3)"", "(1,2)(3,4)"\])
```

(continues on next page)
This group has a variety of interesting properties, so it is worth experimenting with. You may also know it as the “alternating group on 4 symbols,” which Sage will create with the command `AlternatingGroup(4)`.

### Symmetries of a cube

Label vertices of one face of a cube with 1, 2, 3 and 4, and on the opposite face label the vertices 5, 6, 7 and 8 (5 opposite 1, 6 opposite 2, etc.). Consider three axes that run from the center of a face to the center of the opposite face, and consider a quarter-turn rotation about each axis. These three rotations will construct the entire symmetry group. Use

```python
sage: cube = PermutationGroup(['(3,2,6,7)(4,1,5,8)',
                      ....: '(1,2,6,5)(4,3,7,8)', '(1,2,3,4)(5,6,7,8)'])
sage: cube.list()
[(),
 (1,3)(2,4)(5,7)(6,8),
 (1,6)(2,5)(3,8)(4,7),
 (1,8)(2,7)(3,6)(4,5),
 (1,4,3,2)(5,8,7,6),
 (1,2,3,4)(5,6,7,8),
 (1,5)(2,8)(3,7)(4,6),
 (1,7)(2,6)(3,5)(4,8),
 (2,5,4)(3,6,8),
 (1,3,8)(2,7,5),
 (1,6,3)(4,5,7),
 (1,8,6)(2,4,7),
 (1,4)(2,8)(3,5)(6,7),
 (1,2,6,5)(3,7,8,4),
 (1,5,6,2)(3,4,8,7),
 (1,7)(2,3)(4,6)(5,8),
 (2,4,5)(3,8,6),
 (1,3,6)(4,7,5),
 (1,6,8)(2,7,4),
 (1,8,3)(2,5,7),
 (1,4,8,5)(2,3,7,6),
 (1,2)(3,5)(4,6)(7,8),
 (1,5,8,4)(2,6,7,3),
 (1,7)(2,8)(3,4)(5,6)]
```

A cube has four distinct diagonals (joining opposite vertices through the center of the cube). Each symmetry of the cube will cause the diagonals to arrange differently. In this way, we can view an element of the symmetry group as a permutation of four “symbols”—the diagonals. It happens that each of the 24 permutations of the diagonals is created by exactly one symmetry of the 8 vertices of the cube. So this subgroup of $S_8$ is “the same as” $S_4$. In Sage:

```python
sage: cube = PermutationGroup(['(3,2,6,7)(4,1,5,8)',
                      ....: '(1,2,6,5)(4,3,7,8)', '(1,2,3,4)(5,6,7,8)'])

sage: cube.is_isomorphic(SymmetricGroup(4))
True
```

will test to see if the group of symmetries of the cube are “the same as” $S_4$ and so will return True.
Here is another way to create the symmetries of a cube. Number the six faces of the cube as follows: 1 on top, 2 on the bottom, 3 in front, 4 on the right, 5 in back, 6 on the left. Now the same rotations as before (quarter-turns about axes through the centers of two opposite faces) can be used as generators of the symmetry group:

```
sage: cubeface = PermutationGroup(['(1,3,2,5)', '(1,4,2,6)', '(3,4,5,6)'])
sage: cubeface.list()
[(),
 (3,5)(4,6),
 (1,6,5)(2,4,3),
 (1,6,3)(2,4,5),
 (1,5,6)(2,3,4),
 (1,5,4)(2,3,6),
 (1,2)(4,6),
 (1,2)(3,5),
 (1,4,5)(2,6,3),
 (1,4,3)(2,6,5),
 (1,3,4)(2,5,6),
 (1,3,6)(2,5,4),
 (1,4,2,6),
 (1,6)(2,4)(3,5),
 (1,3,2,5),
 (1,5,2,3),
 (1,2)(3,4)(5,6),
 (3,6,5,4),
 (1,6,2,4),
 (1,4)(2,6)(3,5),
 (1,5)(2,3)(4,6),
 (1,3)(2,5)(4,6),
 (3,4,5,6),
 (1,2)(3,6)(4,5)]
```

Again, this subgroup of $S_6$ is “same as” the full symmetric group, $S_4$:

```
sage: cubeface = PermutationGroup(['(1,3,2,5)', '(1,4,2,6)', '(3,4,5,6)'])
sage: cubeface.is_isomorphic(SymmetricGroup(4))
True
```

It turns out that in each of the above constructions, it is sufficient to use just two of the three generators (any two). But one generator is not enough. Give it a try and use Sage to convince yourself that a generator can be sacrificed in each case.

**Normal subgroups**

**Checking normality**

The code below:

1. begins with the alternating group $A_4$;
2. specifies three elements of the group (the three symmetries of the tetrahedron that are 180 degree rotations about axes through midpoints of opposite edges);
3. uses these three elements to generate a subgroup; and finally
4. illustrates the command for testing if the subgroup $H$ is a normal subgroup of the group $A_4$. 

12.1. Thematic tutorial document tree
Thematic Tutorials, Release 9.7

```
sage: A4 = AlternatingGroup(4)
sage: r1 = A4("(1,2) (3,4)")
sage: r2 = A4("(1,3) (2,4)")
sage: r3 = A4("(1,4) (2,3)")
sage: H = A4.subgroup([r1, r2, r3])
sage: H.is_normal(A4)
True
```

**Quotient group**

Extending the previous example, we can create the quotient (factor) group of $A_4$ by $H$. The commands

```
sage: A4 = AlternatingGroup(4)
sage: r1 = A4("(1,2) (3,4)")
sage: r2 = A4("(1,3) (2,4)")
sage: r3 = A4("(1,4) (2,3)")
sage: H = A4.subgroup([r1, r2, r3])
sage: A4.quotient(H)
Permutation Group with generators [(1,2,3)]
```

returns a permutation group generated by $(1,2,3)$. As expected this is a group of order 3. Notice that we do not get back a group of the actual cosets, but instead we get a group isomorphic to the factor group.

**Simple groups**

It is easy to check to see if a group is void of any normal subgroups. The commands

```
sage: AlternatingGroup(5).is_simple()
True
sage: AlternatingGroup(4).is_simple()
False
```

prints True and then False.

**Composition series**

For any group, it is easy to obtain a composition series. There is an element of randomness in the algorithm, so you may not always get the same results. (But the list of factor groups is unique, according to the Jordan-Hölder theorem.) Also, the subgroups generated sometimes have more generators than necessary, so you might want to “study” each subgroup carefully by checking properties like its order.

An interesting example is:

```
DihedralGroup(105).composition_series()
```

The output will be a list of 5 subgroups of $D_{105}$, each a normal subgroup of its predecessor.

Several other series are possible, such as the derived series. Use tab-completion to see the possibilities.
Thematic Tutorials, Release 9.7

Conjugacy
Given a group 𝐺, we can define a relation ∼ on 𝐺 by: for 𝑎, 𝑏 ∈ 𝐺, 𝑎 ∼ 𝑏 if and only if there exists an element 𝑔 ∈ 𝐺
such that 𝑔𝑎𝑔 −1 = 𝑏.
Since this is an equivalence relation, there is an associated partition of the elements of 𝐺 into equivalence
classes. For this very important relation, the classes are known as “conjugacy classes.” A representative of
each of these equivalence classes can be found as follows. Suppose G is a permutation group, then G.
conjugacy_classes_representatives() will return a list of elements of $G$, one per conjugacy class.
Given an element 𝑔 ∈ 𝐺, the “centralizer” of 𝑔 is the set 𝐶(𝑔) = {ℎ ∈ 𝐺 | ℎ𝑔ℎ−1 = 𝑔}, which is a subgroup of 𝐺.
A theorem tells us that the size of each conjugacy class is the order of the group divided by the order of the centralizer
of an element of the class. With the following code we can determine the size of the conjugacy classes of the full
symmetric group on 5 symbols:
sage: G = SymmetricGroup(5)
sage: group_order = G.order()
sage: reps = G.conjugacy_classes_representatives()
sage: class_sizes = []
sage: for g in reps:
....:
class_sizes.append(group_order / G.centralizer(g).order())
...
sage: class_sizes
[1, 10, 15, 20, 20, 30, 24]
This should produce the list [1, 10, 15, 20, 20, 30, 24] which you can check sums to 120, the order of the
group. You might be able to produce this list by counting elements of the group 𝑆5 with identical cycle structure
(which will require a few simple combinatorial arguments).
Sylow subgroups
Sylow’s Theorems assert the existence of certain subgroups. For example, if 𝑝 is a prime, and 𝑝𝑟 divides the order
of a group 𝐺, then 𝐺 must have a subgroup of order 𝑝𝑟 . Such a subgroup could be found among the output of the
conjugacy_classes_subgroups() command by checking the orders of the subgroups produced. The map() command is a quick way to do this. The symmetric group on 7 symbols, 𝑆7 , has order 7! = 5040 and is divisible by
24 = 16. Let’s find one example of a subgroup of permutations on 4 symbols with order 16:
sage: G = SymmetricGroup(7)
sage: subgroups = G.conjugacy_classes_subgroups()
sage: list(map(order, subgroups))
[1, 2, 2, 2, 3, 3, 4, 4, 4, 4, 4, 4, 4, 5, 6, 6, 6, 6, 6, 6, 6, 6, 7, 8, 8, 8, 8, 8, 8,␣
˓→8, 9, 10, 10, 10, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 14, 16, 18, 18,␣
˓→18, 20, 20, 20, 21, 24, 24, 24, 24, 24, 24, 24, 24, 24, 24, 24, 24, 24, 24, 36, 36, 36,
˓→ 36, 40, 42, 48, 48, 48, 60, 60, 72, 72, 72, 72, 120, 120, 120, 120, 144, 168, 240,␣
˓→360, 720, 2520, 5040]
The map(order, subgroups) command will apply the order() function to each of the subgroups in the list
subgroups. The output is thus a large list of the orders of many subgroups (96 to be precise). If you count carefully, you will see that the 49-th subgroup has order 16. You can retrieve this group for further study by referencing it
as subgroups[48] (remember that counting starts at zero).
If 𝑝𝑟 is the highest power of 𝑝 to divide the order of 𝐺, then a subgroup of order 𝑝𝑟 is known as a “Sylow 𝑝-subgroup.” Sylow’s Theorems also say any two Sylow 𝑝-subgroups are conjugate, so the output of
conjugacy_classes_subgroups() should only contain each Sylow 𝑝-subgroup once. But there is an easier way,

12.1. Thematic tutorial document tree

127


sylo_subgroup(p) will return one. Notice that the argument of the command is just the prime $p$, not the full power $p^r$. Failure to use a prime will generate an informative error message.

Groups of small order as permutation groups

We list here constructions, as permutation groups, for all of the groups of order less than 16.

<table>
<thead>
<tr>
<th>Size</th>
<th>Construction</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SymmetricGroup(1)</td>
<td>Trivial</td>
</tr>
<tr>
<td>2</td>
<td>SymmetricGroup(2)</td>
<td>Also CyclicPermutationGroup(2)</td>
</tr>
<tr>
<td>3</td>
<td>CyclicPermutationGroup(3)</td>
<td>Prime order</td>
</tr>
<tr>
<td>4</td>
<td>CyclicPermutationGroup(4)</td>
<td>Cyclic</td>
</tr>
<tr>
<td>5</td>
<td>KleinFourGroup()</td>
<td>Abelian, non-cyclic</td>
</tr>
<tr>
<td>6</td>
<td>CyclicPermutationGroup(6)</td>
<td>Cyclic</td>
</tr>
<tr>
<td>7</td>
<td>CyclicPermutationGroup(7)</td>
<td>Prime order</td>
</tr>
<tr>
<td>8</td>
<td>CyclicPermutationGroup(8)</td>
<td>Cyclic</td>
</tr>
</tbody>
</table>
| 8    | $D_1 = \text{CyclicPermutationGroup}(4)$
$D_2 = \text{CyclicPermutationGroup}(2)$
$G = \text{direct_product_permgroups}([D_1,D_2])$ | Abelian, non-cyclic |
| 8    | $D_1 = \text{CyclicPermutationGroup}(2)$
$D_2 = \text{CyclicPermutationGroup}(2)$
$D_3 = \text{CyclicPermutationGroup}(2)$
$G = \text{direct_product_permgroups}([D_1,D_2,D_3])$ | Abelian, non-cyclic |
| 8    | DihedralGroup(4) | Non-abelian |
| 8    | QuaternionGroup() | Quaternions, also DiCyclicGroup(2) |
| 9    | CyclicPermutationGroup(9) | Cyclic |
| 9    | $D_1 = \text{CyclicPermutationGroup}(3)$
$D_2 = \text{CyclicPermutationGroup}(3)$
$G = \text{direct_product_permgroups}([D_1,D_2])$ | Abelian, non-cyclic |
| 10   | CyclicPermutationGroup(10) | Cyclic |
| 10   | DihedralGroup(5) | Non-abelian |
| 11   | CyclicPermutationGroup(11) | Prime order |
| 12   | CyclicPermutationGroup(12) | Cyclic |
| 12   | $D_1 = \text{CyclicPermutationGroup}(6)$
$D_2 = \text{CyclicPermutationGroup}(2)$
$G = \text{direct_product_permgroups}([D_1,D_2])$ | Abelian, non-cyclic |
| 12   | DihedralGroup(6) | Non-abelian |
| 12   | AlternatingGroup(4) | Non-abelian, symmetries of tetrahedron |
| 12   | DiCyclicGroup(3) | Non-abelian |
|      | $Z_3 \rtimes Z_4$ | Also semi-direct product $Z_3 \rtimes Z_4$ |
| 13   | CyclicPermutationGroup(13) | Prime order |
| 14   | CyclicPermutationGroup(14) | Cyclic |
| 14   | DihedralGroup(7) | Non-abelian |
| 15   | CyclicPermutationGroup(15) | Cyclic |
Acknowledgements

The construction of Sage is the work of many people, and the group theory portion is made possible by the extensive work of the creators of GAP. However, we will single out three people from the Sage team to thank for major contributions toward bringing you the group theory portion of Sage: David Joyner, William Stein, and Robert Bradshaw. Thanks!

12.1.4 Lie Methods and Related Combinatorics in Sage

Author: Daniel Bump (Stanford University), Ben Salisbury (Central Michigan University), and Anne Schilling (UC Davis)

These notes explain how to use the mathematical software Sage for Lie group computations. Sage also contains many combinatorial algorithms. We will cover only some of these.

The Scope of this Document

Lie groups and algebras

Sage can be used to do standard computations for Lie groups and Lie algebras. The following categories of representations are equivalent:

- Complex representations of a compact, semisimple simply connected Lie group $G$.
- Complex representations of its Lie algebra $\mathfrak{g}$. This is a real Lie algebra, so representations are not required to be complex linear maps.
- Complex representations of its complexified Lie algebra $\mathfrak{g}_C = \mathbb{C} \otimes \mathfrak{g}$. This is a complex Lie algebra and representations are required to be complex linear transformations.
- The complex analytic representations of the semisimple simply-connected complex analytic group $G_C$ having $\mathfrak{g}_C$ as its Lie algebra.
- Modules of the universal enveloping algebra $U(\mathfrak{g}_C)$.
- Modules of the quantized enveloping algebra $U_q(\mathfrak{g}_C)$.

For example, we could take $G = SU(n)$, $\mathfrak{g} = \mathfrak{sl}(n, \mathbb{R})$, $\mathfrak{g}_C = \mathfrak{sl}(n, \mathbb{C})$ and $G_C = SL(n, \mathbb{C})$. Because these categories are the same, their representations may be studied simultaneously. The above equivalences may be expanded to include reductive groups like $U(n)$ and $GL(n)$ with a bit of care.

Here are some typical problems that can be solved using Sage:

- Decompose a module in any one of these categories into irreducibles.
- Compute the Frobenius-Schur indicator of an irreducible module.
- Compute the tensor product of two modules.
- If $H$ is a subgroup of $G$, study the restriction of modules for $G$ to $H$. The solution to this problem is called a branching rule.
- Find the multiplicities of the weights of the representation.

In addition to its representations, which we may study as above, a Lie group has various related structures. These include:

- The Weyl Group $W$.
- The Weight Lattice.
• The Root System
• The Cartan Type.
• The Dynkin diagram.
• The extended Dynkin diagram.

Sage contains methods for working with these structures.

If there is something you need that is not implemented, getting it added to Sage will likely be possible. You may write your own algorithm for an unimplemented task, and if it is something others will be interested in, it is probably possible to get it added to Sage.

**Combinatorics**

Sage supports a great many related mathematical objects. Some of these properly belong to combinatorics. It is beyond the scope of these notes to cover all the combinatorics in Sage, but we will try to touch on those combinatorial methods which have some connection with Lie groups and representation theory. These include:

• The affine Weyl group, an infinite group containing \( W \).
• Kashiwara crystals, which are combinatorial analogs of modules in the above categories.
• Coxeter group methods applicable to Weyl groups and the affine Weyl group, such as Bruhat order.
• The Iwahori Hecke algebras, which are deformations of the group algebras of \( W \) and the affine Weyl group.
• Kazhdan-Lusztig polynomials.

**Lie Group Basics**

**Goals of this section**

Since we must be brief here, this is not really a place to learn about Lie groups or Lie algebras. Rather, the point of this section is to outline what you need to know to use Sage effectively for Lie computations, and to fix ideas and notations.

**Semisimple and reductive groups**

If \( g \in GL(n, \mathbb{C}) \), then \( g \) may be uniquely factored as \( g_1g_2 \) where \( g_1 \) and \( g_2 \) commute, with \( g_1 \) semisimple (diagonalizable) and \( g_2 \) unipotent (all its eigenvalues equal to 1). This follows from the Jordan canonical form. If \( g = g_1 \) then \( g \) is called *semisimple* and if \( g = g_2 \) then \( g \) is called *unipotent*.

We consider a Lie group \( G \) and a class of representations such that if an element \( g \in G \) is unipotent (resp. semisimple) in one faithful representation from the class, then it is unipotent (resp. semisimple) in every faithful representation of the class. Thus the notion of being semisimple or unipotent is intrinsic. Examples:

• Compact Lie groups with continuous representations
• Complex analytic groups with analytic representations
• Algebraic groups over \( \mathbb{R} \) with algebraic representations.

A subgroup of \( G \) is called *unipotent* if it is connected and all its elements are unipotent. It is called a *torus* if it is connected, abelian, and all its elements are semisimple. The group \( G \) is called *reductive* if it has no nontrivial normal unipotent subgroup. For example, \( GL(2, \mathbb{C}) \) is reductive, but its subgroup:

\[
\left\{ \begin{pmatrix} a & b \\ d & c \end{pmatrix} \right\}
\]
is not since it has a normal unipotent subgroup
\[
\left\{ \begin{pmatrix} 1 & b \\ 0 & 1 \end{pmatrix} \right\}.
\]
A group has a unique largest normal unipotent subgroup, called the unipotent radical, so it is reductive if and only if the unipotent radical is trivial.

A Lie group is called semisimple if it is reductive and furthermore has no nontrivial normal tori. For example $GL(2, \mathbb{C})$ is reductive but not semisimple because it has a normal torus:
\[
\left\{ \begin{pmatrix} a & \ast \\ \ast & a \end{pmatrix} \right\}.
\]
The group $SL(2, \mathbb{C})$ is semisimple.

### Fundamental group and center

If $G$ is a semisimple Lie group then its center and fundamental group are finite abelian groups. The universal covering group $\tilde{G}$ is therefore a finite extension with the same Lie algebra. Any representation of $G$ may be reinterpreted as a representation of the simply connected $G$. Therefore we may as well consider representations of $\tilde{G}$ and restrict ourselves to the simply connected group.

### Parabolic subgroups and Levi subgroups

Let $G$ be a reductive complex analytic group. A maximal solvable subgroup of $G$ is called a Borel subgroup. All Borel subgroups are conjugate. Any subgroup $P$ containing a Borel subgroup is called a parabolic subgroup. We may write $P$ as a semidirect product of its maximal normal unipotent subgroup or unipotent radical $P$ and a reductive subgroup $M$, which is determined up to conjugacy. The subgroup $M$ is called a Levi subgroup.

**Example:** Let $G = GL_n(\mathbb{C})$ and let $r_1, \ldots, r_k$ be integers whose sum is $n$. Then we may consider matrices of the form:
\[
\begin{pmatrix}
g_1 & * & \cdots & * \\
g_2 & * & & \\
& \ddots & \ddots & \\
g_r & & & 
\end{pmatrix},
\]
where $g_i \in GL(r_i, \mathbb{C})$. The unipotent radical consists of the subgroup in which all $g_i = I_{r_i}$. The Levi subgroup (determined up to conjugacy) is:
\[
M = \left\{ \begin{pmatrix} g_1 \\ g_2 \\ \vdots \\ g_r \end{pmatrix} \right\},
\]
and is isomorphic to $M = GL(r_1, \mathbb{C}) \times \cdots \times GL(r_k, \mathbb{C})$. Therefore $M$ is a Levi subgroup.

The notion of a Levi subgroup can be extended to compact Lie groups. Thus $U(r_1) \times \cdots \times U(r_k)$ is a Levi subgroup of $U(n)$. However parabolic subgroups do not exist for compact Lie groups.
Cartan types

Semisimple Lie groups are classified by their Cartan types. There are both reducible and irreducible Cartan types in Sage. Let us start with the irreducible types. Such a type is implemented in Sage as a pair ['X', r] where ‘X’ is one of A, B, C, D, E, F or G and r is a positive integer. If ‘X’ is ‘D’ then we must have r > 1 and if ‘X’ is one of the exceptional types ‘E’, ‘F’ or ‘G’ then r is limited to only a few possibilities. The exceptional types are:

['G', 2], ['F', 4], ['E', 6], ['E', 7] or ['E', 8].

A simply-connected semisimple group is a direct product of simple Lie groups, which are given by the following table. The integer r is called the rank, and is the dimension of the maximal torus.

<table>
<thead>
<tr>
<th>compact group</th>
<th>complex analytic group</th>
<th>Cartan type</th>
</tr>
</thead>
<tbody>
<tr>
<td>(SL(r+1))</td>
<td>(SL(r+1, \mathbb{C}))</td>
<td>(A_r)</td>
</tr>
<tr>
<td>(Spin(2r+1))</td>
<td>(Spin(2r+1, \mathbb{C}))</td>
<td>(B_r)</td>
</tr>
<tr>
<td>(Sp(2r))</td>
<td>(Sp(2r, \mathbb{C}))</td>
<td>(C_r)</td>
</tr>
<tr>
<td>(Spin(2r))</td>
<td>(Spin(2r, \mathbb{C}))</td>
<td>(D_r)</td>
</tr>
</tbody>
</table>

You may create these Cartan types and their Dynkin diagrams as follows:

```
sage: ct = CartanType("D5"); ct[['D', 5]
```

Here "D5" is an abbreviation for ['D', 5]. The group \(Spin(n)\) is the simply-connected double cover of the orthogonal group \(SO(n)\).

Dual Cartan types

Every Cartan type has a dual, which you can get from within Sage:

```
sage: CartanType("B4").dual()[['C', 4]
```

Types other than \(B_r\) and \(C_r\) for \(r > 2\) are self-dual in the sense that the dual is isomorphic to the original type; however the isomorphism of a Cartan type with its dual might relabel the vertices. We can see this as follows:

```
sage: CartanType("F4").dynkin_diagram() |
0---0=>=0---0
|  1 2 3 4
F4
```
sage: CartanType("F4").dual()[['F', 4] relabelled by {1: 4, 2: 3, 3: 2, 4: 1}
sage: CartanType("F4").dual().dynkin_diagram() |
0---0=>=0---0
|  4 3 2 1
F4 relabelled by {1: 4, 2: 3, 3: 2, 4: 1}
Reducible Cartan types

If \( G \) is a Lie group of finite index in \( G_1 \times G_2 \), where \( G_1 \) and \( G_2 \) are Lie groups of positive dimension, then \( G \) is called reducible. In this case, the root system of \( G \) is the disjoint union of the root systems of \( G_1 \) and \( G_2 \), which lie in orthogonal subspaces of the ambient space of the weight space of \( G \). The Cartan type of \( G \) is thus reducible.

Reducible Cartan types are supported in Sage as follows:

```sage
sage: RootSystem("A1xA1")
Root system of type A1xA1
sage: WeylCharacterRing("A1xA1")
The Weyl Character Ring of Type A1xA1 with Integer Ring coefficients
```

Low dimensional Cartan types

There are some isomorphisms that occur in low degree.

<table>
<thead>
<tr>
<th>Cartan Type</th>
<th>Group</th>
<th>Equivalent Type</th>
<th>Isomorphic Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>( B_2 )</td>
<td>spin(5)</td>
<td>( C_2 )</td>
<td>( Sp(4) )</td>
</tr>
<tr>
<td>( D_3 )</td>
<td>spin(6)</td>
<td>( A_3 )</td>
<td>( SL(4) )</td>
</tr>
<tr>
<td>( D_2 )</td>
<td>spin(4)</td>
<td>( A_1 \times A_1 )</td>
<td>( SL(2) \times SL(2) )</td>
</tr>
<tr>
<td>( B_1 )</td>
<td>spin(3)</td>
<td>( A_1 )</td>
<td>( SL(2) )</td>
</tr>
<tr>
<td>( C_1 )</td>
<td>( Sp(2) )</td>
<td>( A_1 )</td>
<td>( SL(2) )</td>
</tr>
</tbody>
</table>

Sometimes the redundant Cartan types such as \( D_3 \) and \( D_2 \) are excluded from the list of Cartan types. However Sage allows them since excluding them leads to exceptions having to be made in algorithms. A better approach, which is followed by Sage, is to allow the redundant Cartan types, but to implement the isomorphisms explicitly as special cases of branching rules. The utility of this approach may be seen by considering that the rank one group \( SL(2) \) has different natural weight lattices realizations depending on whether we consider it to be \( SL(2) \), \( spin(2) \) or \( Sp(2) \):

```sage
sage: RootSystem("A1").ambient_space().simple_roots()
Finite family {1: (1, -1)}
sage: RootSystem("B1").ambient_space().simple_roots()
Finite family {1: (1)}
sage: RootSystem("C1").ambient_space().simple_roots()
Finite family {1: (2)}
```

Relabeled Cartan types

By default Sage uses the labeling of the Dynkin diagram from [Bourbaki46]. There is another labeling of the vertices due to Dynkin. Most of the literature follows [Bourbaki46], though [Kac] follows Dynkin.

If you need to use Dynkin’s labeling, you should be aware that Sage does support relabeled Cartan types. See the documentation in `sage.combinat.root_system.type_relabel` for further information.
Standard realizations of the ambient spaces

These realizations follow the Appendix in [Bourbaki46]. See the Root system plot tutorial for how to visualize them.

Type A

For type $A_r$ we use an $r + 1$ dimensional ambient space. This means that we are modeling the Lie group $U(r + 1)$ or $GL(r + 1, \mathbb{C})$ rather than $SU(r + 1)$ or $SL(r + 1, \mathbb{C})$. The ambient space is identified with $\mathbb{Q}^{r+1}$:

\begin{verbatim}
sage: RootSystem("A3").ambient_space().simple_roots()
Finite family {1: (1, -1, 0, 0), 2: (0, 1, -1, 0), 3: (0, 0, 1, -1)}
sage: RootSystem("A3").ambient_space().fundamental_weights()
Finite family {1: (1, 0, 0, 0), 2: (1, 1, 0, 0), 3: (1, 1, 1, 0)}
sage: RootSystem("A3").ambient_space().rho()
(3, 2, 1, 0)
\end{verbatim}

The dominant weights consist of integer $r + 1$-tuples $\lambda = (\lambda_1, \ldots, \lambda_{r+1})$ such that $\lambda_1 \geq \cdots \geq \lambda_{r+1}$.

See SL versus GL for further remarks about Type A.

Type B

For the remaining classical Cartan types $B_r$, $C_r$ and $D_r$ we use an $r$-dimensional ambient space:

\begin{verbatim}
sage: RootSystem("B3").ambient_space().simple_roots()
Finite family {1: (1, -1, 0), 2: (0, 1, -1), 3: (0, 0, 1)}
sage: RootSystem("B3").ambient_space().fundamental_weights()
Finite family {1: (1, 0, 0), 2: (1, 1, 0), 3: (1/2, 1/2, 1/2)}
sage: RootSystem("B3").ambient_space().rho()
(5/2, 3/2, 1/2)
\end{verbatim}

This is the Cartan type of $spin(2r+1)$. The last fundamental weight $(1/2, 1/2, \ldots, 1/2)$ is the highest weight of the $2^r$-dimensional spin representation. All the other fundamental representations factor through the homomorphism $spin(2r+1) \rightarrow SO(2r+1)$ and are representations of the orthogonal group.

The dominant weights consist of $r$-tuples of integers or half-integers $(\lambda_1, \ldots, \lambda_r)$ such that $\lambda_1 \geq \lambda_2 \cdots \geq \lambda_r \geq 0$, and such that the differences $\lambda_i - \lambda_j \in \mathbb{Z}$.

Type C

\begin{verbatim}
sage: RootSystem("C3").ambient_space().simple_roots()
Finite family {1: (1, -1, 0), 2: (0, 1, -1), 3: (0, 0, 2)}
sage: RootSystem("C3").ambient_space().fundamental_weights()
Finite family {1: (1, 0, 0), 2: (1, 1, 0), 3: (1, 1, 1)}
sage: RootSystem("C3").ambient_space().rho()
(3, 2, 1)
\end{verbatim}

This is the Cartan type of the symplectic group $Sp(2r)$.

The dominant weights consist of $r$-tuples of integers $\lambda = (\lambda_1, \ldots, \lambda_{r+1})$ such that $\lambda_1 \geq \cdots \geq \lambda_r \geq 0$. 

134 Chapter 12. Documentation
Type D

```
sage: RootSystem("D4").ambient_space().simple_roots()
Finite family {1: (1, -1, 0, 0), 2: (0, 1, -1, 0), 3: (0, 0, 1, -1), 4: (0, 0, 1, 1)}
sage: RootSystem("D4").ambient_space().fundamental_weights()
Finite family {1: (1, 0, 0, 0), 2: (1, 1, 0, 0), 3: (1/2, 1/2, 1/2, -1/2), 4: (1/2, 1/2, 1/2, 1/2)}
sage: RootSystem("D4").ambient_space().rho()
(3, 2, 1, 0)
```

This is the Cartan type of \(\text{spin}(2r)\). The last two fundamental weights are the highest weights of the two \(2^{r-1}\)-dimensional spin representations.

The dominant weights consist of \(r\)-tuples of integers \(\lambda = (\lambda_1, \ldots, \lambda_{r+1})\) such that \(\lambda_1 \geq \cdots \geq \lambda_{r-1} \geq |\lambda_r|\).

**Exceptional Types**

We leave the reader to examine the exceptional types. You can use Sage to list the fundamental dominant weights and simple roots.

**Weights and the ambient space**

Let \(G\) be a reductive complex analytic group. Let \(T\) be a maximal torus, \(\Lambda = X^*(T)\) be its group of analytic characters. Then \(T \cong (C^*)^r\) for some \(r\) and \(\Lambda \cong \mathbb{Z}^r\).

**Example 1:** Let \(G = \text{GL}_{r+1}(C)\). Then \(T\) is the diagonal subgroup and \(X^*(T) \cong \mathbb{Z}^{r+1}\). If \(\lambda = (\lambda_1, \ldots, \lambda_n)\) then \(\lambda\) is identified with the rational character

\[
\begin{pmatrix}
t_1 \\
\vdots \\
t_n
\end{pmatrix} \mapsto \prod t_i^{\lambda_i}.
\]

**Example 2:** Let \(G = \text{SL}_{r+1}(C)\). Again \(T\) is the diagonal subgroup but now if \(\lambda \in \mathbb{Z}^\Delta = \{(d, \cdots, d) | d \in \mathbb{Z}\} \subseteq \mathbb{Z}^{r+1}\) then \(\prod t_i^{\lambda_i} = \det(t)^d = 1\), so \(X^*(T) \cong \mathbb{Z}^{r+1}/\mathbb{Z}^\Delta \cong \mathbb{Z}^r\).

- Elements of \(\Lambda\) are called *weights*.
- If \(\pi : G \to GL(V)\) is any representation we may restrict \(\pi\) to \(T\). Then the characters of \(T\) that occur in this restriction are called the *weights* of \(\pi\).
- \(G\) acts on its Lie algebra by conjugation (the *adjoint representation*).
- The nonzero weights of the adjoint representation are called *roots*.
- The *ambient space* of \(\Lambda\) is \(Q \otimes \Lambda\).
The root system

As we have mentioned, $G$ acts on its complexified Lie algebra $\mathfrak{g}_C$ by the adjoint representation. The zero weight space $\mathfrak{g}_C(0)$ is just the Lie algebra of $T$ itself. The other nonzero weights each appear with multiplicity one and form an interesting configuration of vectors called the root system $\Phi$.

It is convenient to partition $\Phi$ into two sets $\Phi^+$ and $\Phi^-$ such that $\Phi^+$ consists of all roots lying on one side of a hyperplane. Often we arrange things so that $G$ is embedded in $GL(n, \mathbb{C})$ in such a way that the positive weights correspond to upper triangular matrices. Thus if $\alpha$ is a positive root, its weight space $\mathfrak{g}_C(\alpha)$ is spanned by a vector $X_\alpha$, and the exponential of this eigenspace in $G$ is a one-parameter subgroup of unipotent matrices. It is always possible to arrange that this one-parameter subgroup consists of upper triangular matrices.

If $\alpha$ is a positive root that cannot be decomposed as a sum of other positive roots, then $\alpha$ is called a simple root. If $G$ is semisimple of rank $r$, then $r$ is the number of positive roots. Let $\alpha_1, \ldots, \alpha_r$ be these.

The Weyl group

Let $G$ be a complex analytic group. Let $T$ be a maximal torus, and let $N(T)$ be its normalizer. Let $W = N(T)/T$ be the Weyl group. It acts on $T$ by conjugation; therefore it acts on the weight lattice $\Lambda$ and its ambient space. The ambient space admits an inner product that is invariant under this action. Let $(v|w)$ denote this inner product. If $\alpha$ is a root let $r_\alpha$ denote the reflection in the hyperplane of the ambient space that is perpendicular to $\alpha$. If $\alpha = \alpha_i$ is a simple root, then we use the notation $s_i$ to denote $r_\alpha$.

Then $s_1, \ldots, s_r$ generate $W$, which is a Coxeter group. This means that it is generated by elements $s_i$ of order two and that if $m(i,j)$ is the order of $s_i s_j$, then

$$W = \left\langle s_i \mid s_i^2 = 1, (s_i s_j)^{m(i,j)} = 1 \right\rangle$$

is a presentation. An important function $\ell : W \to \mathbb{Z}$ is the length function, where $\ell(w)$ is the length of the shortest decomposition of $w$ into a product of simple reflections.

The dual root system

The coroots are certain linear functionals on the ambient space that also form a root system. Since the ambient space admits a $W$-invariant inner product $(\cdot | \cdot)$, they may be identified with elements of the ambient space itself. Then they are proportional to the roots, though if the roots have different lengths, long roots correspond to short coroots and conversely. The coroot corresponding to the root $\alpha$ is

$$\alpha^\vee = \frac{2\alpha}{(\alpha|\alpha)}.$$ 

We can also describe the natural pairing between coroots and roots using this invariant inner product as

$$\langle \alpha^\vee, \beta \rangle = 2 \frac{(\alpha|\beta)}{(\alpha|\alpha)}.$$
The Dynkin diagram

The Dynkin diagram is a graph whose vertices are in bijection with the set simple roots. We connect the vertices corresponding to roots that are not orthogonal. Usually two such roots (vertices) make an angle of $2\pi/3$, in which case we connect them with a single bond. Occasionally they may make an angle of $3\pi/4$ in which case we connect them with a double bond, or $5\pi/6$ in which case we connect them with a triple bond. If the bond is single, the roots have the same length with respect to the inner product on the ambient space. In the case of a double or triple bond, the two simple roots in question have different length, and the bond is drawn as an arrow from the long root to the short root. Only the exceptional group $G_2$ has a triple bond.

There are various ways to get the Dynkin diagram in Sage:

```
sage: DynkinDiagram("D5")
    O 5
      |
    O---O---O---O
    1 2 3 4
D5
sage: ct = CartanType("E6"); ct
['E', 6]
sage: ct.dynkin_diagram()
    O 2
      |
    O---O---O---O---O
    1 3 4 5 6
E6
sage: B4 = WeylCharacterRing("B4"); B4
The Weyl Character Ring of Type B4 with Integer Ring coefficients
sage: B4.dynkin_diagram()
    O---O---O=>=O
    1 2 3 4
B4
sage: RootSystem("G2").dynkin_diagram()
    3
    O=<=O
    1 2
G2
```

The Cartan matrix

Consider the natural pairing $\langle \cdot, \cdot \rangle$ between coroots and roots, then the defining matrix of this pairing is called the Cartan matrix. That is to say, the Cartan matrix $A = (a_{ij})_{ij}$ is given by

$$a_{ij} = \langle \alpha_i^\vee, \alpha_j \rangle.$$

This uniquely corresponds to a root system/Dynkin diagram/Lie group.

We note that we have made a convention choice, and the opposite convention corresponds to taking the transpose of the Cartan matrix.
Fundamental weights and the Weyl vector

There are certain weights $\omega_1, \ldots, \omega_r$ that:

$$\langle \omega_j, \alpha_i \rangle = 2 \left( \frac{\alpha_i | \omega_j}{\alpha_i | \alpha_i} \right) = \delta_{ij}.$$ 

If $G$ is semisimple then these are uniquely determined, whereas if $G$ is reductive but not semisimple we may choose them conveniently.

Let $\rho$ be the sum of the fundamental dominant weights. If $G$ is semisimple, then $\rho$ is half the sum of the positive roots. In case $G$ is not semisimple, we have noted, the fundamental weights are not completely determined by the inner product condition given above. If we make a different choice, then $\rho$ is altered by a vector that is orthogonal to all roots. This is a harmless change for many purposes such as the Weyl character formula.

In Sage, this issue arises only for Cartan type $A_r$. See SL versus GL.

Representations and characters

Let $T$ be a maximal torus and $\Lambda = X^*(T)$ be the group of rational characters. Then $\Lambda \cong \mathbb{Z}^r$.

- Recall that elements of $\Lambda \cong \mathbb{Z}^r$ are called weights.
- The Weyl group $W = N(T)/T$ acts on $T$, hence on $\Lambda$ and its ambient space by conjugation.
- The ambient space $\mathbb{Q} \otimes X^*(T) \cong \mathbb{Q}^r$ has a fundamental domain $\mathcal{C}^+$ for the Weyl group $W$ called the positive Weyl chamber. Weights in $\mathcal{C}^+$ are called dominant.
- Then $\mathcal{C}^+$ consists of all vectors such that $(\alpha | v) \geq 0$ for all positive roots $\alpha$.
- It is useful to embed $\Lambda$ in $\mathbb{R}^r$ and consider weights as lattice points.
- If $(\pi, V)$ is a representation then restricting to $T$, the module $V$ decomposes into a direct sum of weight eigenspaces $V(\mu)$ with multiplicity $m(\mu)$ for weight $\mu$.
- There is a unique highest weight $\lambda$ with respect to the partial order. We have $\lambda \in \mathcal{C}$ and $m(\lambda) = 1$.
- $V \leftrightarrow \lambda$ gives a bijection between irreducible representations and weights $\lambda$ in $\mathcal{C}^+$.

Assuming that $G$ is simply-connected (or more generally, reductive with a simply-connected derived group) every dominant weight $\lambda$ is the highest weight of a unique irreducible representation $\pi_{\lambda}$, and $\lambda \mapsto \pi_{\lambda}$ gives a parametrization of the isomorphism classes of irreducible representations of $G$ by the dominant weights.

The character of $\pi_{\lambda}$ is the function $\chi_{\lambda}(g) = tr(\pi_{\lambda}(g))$. It is determined by its values on $T$. If $(z) \in T$ and $\mu \in \Lambda$, let us write $z^\mu$ for the value of $\mu$ on $z$. Then the character:

$$\chi_{\lambda}(z) = \sum_{\mu \in \Lambda} m(\mu) z^\mu.$$ 

Sometimes this is written

$$\chi_{\lambda} = \sum_{\mu \in \Lambda} m(\mu) e^{\lambda}.$$ 

The meaning of $e^{\lambda}$ is subject to interpretation, but we may regard it as the image of the additive group $\Lambda$ in its group algebra. The character is then regarded as an element of this ring, the group algebra of $\Lambda$. 


In this example, \( G = \text{SL}(3, \mathbb{C}) \). We have drawn the weights of an irreducible representation with highest weight \( \lambda \). The shaded region is \( C^+ \). \( \lambda \) is a dominant weight, and the labeled vertices are the weights with positive multiplicity in \( V(\lambda) \). The weights weights on the outside have \( m(\mu) = 1 \), while the six interior weights (with double circles) have \( m(\mu) = 2 \).

**Partitions and Schur polynomials**

The considerations of this section are particular to type \( A \). We review the relationship between characters of \( GL(n, \mathbb{C}) \) and symmetric function theory.

A *partition* \( \lambda \) is a sequence of descending nonnegative integers:

\[
\lambda = (\lambda_1, \lambda_2, \ldots, \lambda_n), \quad \lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n \geq 0.
\]

We do not distinguish between two partitions if they differ only by some trailing zeros, so \( (3, 2) = (3, 2, 0) \). If \( l \) is the last integer such that \( \lambda_l > 0 \) then we say that \( l \) is the *length* of \( \lambda \). If \( k = \sum \lambda_i \) then we say that \( \lambda \) is a *partition* of \( k \) and write \( \lambda \vdash k \).

A partition of length \( \leq n = r + 1 \) is therefore a dominant weight of type \( ['A', r] \). Not every dominant weight is a partition, since the coefficients in a dominant weight could be negative. Let us say that an element \( \mu = (\mu_1, \mu_2, \cdots, \mu_n) \) of the \( ['A', r] \) root lattice is *effective* if the \( \mu_i \geq 0 \). Thus an effective dominant weight of \( ['A', r] \) is a partition of length \( \leq n \), where \( n = r + 1 \).

Let \( \lambda \) be a dominant weight, and let \( \chi_\lambda \) be the character of \( GL(n, \mathbb{C}) \) with highest weight \( \lambda \). If \( k \) is any integer we may consider the weight \( \mu = (\lambda_1 + k, \ldots, \lambda_n + k) \) obtained by adding \( k \) to each entry. Then \( \chi_\mu = \det^k \otimes \chi_\lambda \). Clearly by choosing \( k \) large enough, we may make \( \mu \) effective.

So the characters of irreducible representations of \( GL(n, \mathbb{C}) \) do not all correspond to partitions, but the characters indexed by partitions (effective dominant weights) are enough that we can write any character \( \det^{-k} \chi_\mu \) where \( \mu \) is a partition. If we take \( k = -\lambda_n \) we could also arrange that the last entry in \( \lambda \) is zero.

If \( \lambda \) is an effective dominant weight, then every weight that appears in \( \chi_\lambda \) is effective. (Indeed, it lies in the convex hull
of $w(\lambda)$ where $w$ runs through the Weyl group $W = S_n$.) This means that if
\[
g = \begin{pmatrix} z_1 & & \\ & \ddots & \\ & & z_n \end{pmatrix} \in GL(n, \mathbb{C})
\]
then $\chi_{\lambda}(g)$ is a polynomial in the eigenvalues of $g$. This is the *Schur polynomial* $s_{\lambda}(z_1, \ldots, z_n)$.

## Affine Cartan types

There are also affine Cartan types, which correspond to (infinite dimensional) affine Lie algebras. There are affine Cartan types of the form $[\`X`, r, 1]$ if $X=A,B,C,D,E,F,G$ and $[\`X`, r]$ is an ordinary Cartan type. There are also *twisted affine* types of the form $[X, r, k]$, where $k = 2$ or $3$ if the Dynkin diagram of the ordinary Cartan type $[X, r]$ has an automorphism of degree $k$. When $k = 1$, the affine Cartan type is said to be *untwisted*.

Illustrating some of the methods available for the untwisted affine Cartan type $[\`A`, 4, 1]$:

```
sage: ct = CartanType(['A',4,1]); ct
['A', 4, 1]
sage: ct.dual()
['A', 4, 1]
sage: ct.classical()
['A', 4]
sage: ct.dynkin_diagram()
0
0-----------+
| |
| |
0---0---0---0
1 2 3 4
A4~
```

The twisted affine Cartan types are relabeling of the duals of certain untwisted Cartan types:

```
sage: CartanType(['A',3,2])
['B', 2, 1]^*
sage: CartanType(['D',4,3])
['G', 2, 1] relabelled by {0: 0, 1: 2, 2: 1}
```

### The affine root and the extended Dynkin diagram

For the extended Dynkin diagram, we add one negative root $\alpha_0$. For the untwisted types, this is the root whose negative is the highest weight in the adjoint representation. Sometimes this is called the *affine root*. We make the Dynkin diagram as before by measuring the angles between the roots. This extended Dynkin diagram is useful for many purposes, such as finding maximal subgroups and for describing the affine Weyl group.

In particular, the hyperplane for the reflection $r_0$, used in generating the affine Weyl group, is translated off the origin (so it becomes an affine hyperplane). Now the root system is not described as linear transformations on an Euclidean space, but instead by *affine* transformations. Thus the dominant chamber has finite volume and tiles the Euclidean space. Moreover, each such tile corresponds to a unique element in the affine Weyl group.

The extended Dynkin diagram may be obtained as the Dynkin diagram of the corresponding untwisted affine type:
sage: ct = CartanType("E6"); ct
['E', 6]
sage: ct.affine()
['E', 6, 1]
sage: ct.affine() == CartanType(['E',6,1])
True
sage: ct.affine().dynkin_diagram()
O 0
| |
O---O---O---O---O
1 3 4 5 6
E6~

The extended Dynkin diagram is also a method of the WeylCharacterRing:

sage: WeylCharacterRing("E7").extended_dynkin_diagram()
O 2
| |
O---O---O---O---O---O---O
0 1 3 4 5 6 7
E7~

We note the following important distinctions from the classical cases:

- The affine Weyl groups are all infinite.
- Type $A_1^{(1)}$ has two anti-parallel roots with distinct reflections. The Dynkin diagram in this case is represented by a double bond with arrows going in both directions.

**Twisted affine root systems**

For the construction of $\alpha_0$ in the twisted types, we refer the reader to Chapter 8 of [Kac]. As mentioned above, most twisted types can be constructed by taking the dual root system of an untwisted type. However, the type $A_2^{(2)}$ root system can only be constructed by the twisting procedure defined in [Kac]. It has the following properties:

- The Dynkin diagram of type $A_2^{(2)}$ has a quadruple bond with an arrow pointing from the short root to the long root.
- Type $A_{2n}^{(2)}$ for $n > 1$ has 3 different root lengths.
Further Generalizations

If a root system (on an Euclidean space) has only the angles $\pi/2, 2\pi/3, 3\pi/4, 5\pi/6$ between its roots, then we call the root system crystallographic (on Wikipedia article Root_system, this condition is called integrality since for any two roots we have $(\beta, \alpha) \in \mathbb{Z}$). So if we look at the reflection group generated by the roots (this is not a Weyl group), we get general Coxeter groups (with non-infinite labels) and non-crystallographic Coxeter groups are not connected with Lie theory.

However we can generalize Dynkin diagrams (equivalently Cartan matrices) to have all its edges labelled by $(a, b)$ where $a, b \in \mathbb{Z}_{>0}$ and corresponds to having $a$ arrows point one way and $b$ arrows pointing the other. For example in type $A_1^{(1)}$, we have one edge of $(2, 2)$, or in type $A_2^{(2)}$, we have one edge of $(1, 4)$ (equivalently $(4, 1)$). These edge label between $i$ and $j$ corresponds to the entries $a_{ij}$ and $a_{ji}$ in the Cartan matrix. These are used to construct a class of (generally infinite dimensional) Lie algebras called Kac-Moody (Lie) algebras, which in turn are used to construct quantum groups. We refer the reader to [Kac] and [HongKang2002] for more information.

The Weyl Character Ring

Weyl character rings

The Weyl character ring is the representation ring of a compact Lie group. It has a basis consisting of the irreducible representations of $G$, or equivalently, their characters. The addition and multiplication in the Weyl character ring correspond to direct sum and tensor product of representations.

Methods of the ambient space

In Sage, many useful features of the Lie group are available as methods of the ambient space:

```sage
sage: S = RootSystem("B2").ambient_space(); S
Ambient space of the Root system of type ['B', 2]
sage: S.roots()
[(1, -1), (1, 1), (1, 0), (0, 1), (-1, 1), (-1, -1), (-1, 0), (0, -1)]
sage: S.fundamental_weights()
Finite family {1: (1, 0), 2: (1/2, 1/2)}
sage: S.positive_roots()
[(1, -1), (1, 1), (1, 0)]
sage: S.weyl_group()
Weyl Group of type ['B', 2] (as a matrix group acting on the ambient space)
```

Methods of the Weyl character ring

If you are going to work with representations, you may want to create a Weyl Character ring. Many methods of the ambient space are available as methods of the Weyl character ring:

```sage
sage: B3 = WeylCharacterRing("B3")
sage: B3.fundamental_weights()
Finite family {1: (1, 0, 0), 2: (1, 1, 0), 3: (1/2, 1/2, 1/2)}
sage: B3.simple_roots()
Finite family {1: (1, -1, 0), 2: (0, 1, -1), 3: (0, 0, 1)}
sage: B3.dynkin_diagram()
O---O=>=O
```

(continues on next page)
Other useful methods of the Weyl character ring include:

- `cartan_type`
- `highest_root`
- `positive_root`
- `extended_dynkin_diagram`
- `rank`

Some methods of the ambient space are not implemented as methods of the Weyl character ring. However, the ambient space itself is a method, and so you have access to its methods from the Weyl character ring:

```sage
sage: B3 = WeylCharacterRing("B3")
sage: B3.space().weyl_group()
Weyl Group of type ['B', 3] (as a matrix group acting on the ambient space)
sage: B3.space()
Ambient space of the Root system of type ['B', 3]
sage: B3.space().rho()
(5/2, 3/2, 1/2)
sage: B3.cartan_type()
['B', 3]
```

**Coroot notation**

It is useful to give the Weyl character ring a name that corresponds to its Cartan type. This has the effect that the ring can parse its own output:

```sage
sage: G2 = WeylCharacterRing("G2")
sage: [G2(fw) for fw in G2.fundamental_weights()]
[G2(1,0,-1), G2(2,-1,-1)]
sage: G2(1,0,-1)
G2(1,0,-1)
sage: G2(0,0,1).degree()
8
```

Actually the prefix for the ring is configurable, so you don’t really have to call this ring `G2`. Type `WeylCharacterRing?` at the `sage:` prompt for details.

There is one important option that you may want to know about. This is **coroot notation**. You select this by specifying the option `style="coroots"` when you create the ring. With the coroot style, the fundamental weights are represented \((1,0,0,\ldots),(0,1,0,\ldots)\) instead of as vectors in the ambient space:

```sage
sage: B3 = WeylCharacterRing("B3", style="coroots")
sage: [B3(fw) for fw in B3.fundamental_weights()]
[B3(1,0,0), B3(0,1,0), B3(0,0,1)]
sage: B3(0,0,1)
B3(0,0,1)
sage: B3(0,0,1).degree()
8
```
The last representation is the eight dimensional spin representation of $G = spin(7)$, the double cover of the orthogonal group $SO(7)$. In the default notation it would be represented $B_3(1/2, 1/2, 1/2)$.

With the coroot notation every irreducible representation is represented $B_3(a, b, c)$ where $a$, $b$ and $c$ are nonnegative integers. This is often convenient. For many purposes the coroot style is preferable.

One disadvantage: in the coroot style the Lie group or Lie algebra is treated as semisimple, so you lose the distinction between $GL(n)$ and $SL(n)$; you also some information about representations of $E_6$ and $E_7$ for the same reason.

Tensor products of representations

The multiplication in the Weyl character ring corresponds to tensor product. This gives us a convenient way of decomposing a tensor product into irreducibles:

```python
sage: B3 = WeylCharacterRing("B3")
sage: fw = B3.fundamental_weights()
sage: spinweight = fw[3]; spinweight
(1/2, 1/2, 1/2)
sage: spin = B3(spinweight); spin
B3(1/2,1/2,1/2)
sage: spin.degree()
8
```

The element $spin$ of the WeylCharacterRing is the representation corresponding to the third highest weight representation, the eight-dimensional spin representation of $spin(7)$. We could just as easily construct it with the command:

```python
sage: spin = B3(1/2,1/2,1/2)
```

We may compute its tensor product with itself, using the multiplicative structure of the Weyl character ring:

```python
sage: chi = spin*spin; chi
B3(0,0,0) + B3(1,0,0) + B3(1,1,0) + B3(1,1,1)
```

We have taken the eight-dimensional spin representation and tensored with itself. We see that the tensor square splits into four irreducibles, each with multiplicity one.

The highest weights that appear here are available (with their coefficients) through the usual free module accessors:

```python
sage: from pprint import pprint
sage: list(chi)
# random
[((1, 1, 1), 1), ((0, 0, 0), 1), ((1, 0, 0), 1), ((1, 1, 0), 1)]
sage: sorted(chi, key=str)
[((0, 0, 0), 1), ((1, 0, 0), 1), ((1, 1, 0), 1), ((1, 1, 1), 1)]
sage: pprint(dict(chi))
{(0, 0, 0): 1, (1, 0, 0): 1, (1, 1, 0): 1, (1, 1, 1): 1}
sage: M = sorted(chi.monomials(), key=\lambda x: x.support()); M
[B3(0,0,0), B3(1,0,0), B3(1,1,0), B3(1,1,1)]
sage: sorted(chi.support())
[(0, 0, 0), (1, 0, 0), (1, 1, 0), (1, 1, 1)]
sage: chi.coefficients()
[1, 1, 1, 1]
sage: [r.degree() for r in M]
[1, 7, 21, 35]
sage: sum(r.degree() for r in chi.monomials())
64
```
Here we have extracted the individual representations, computed their degrees and checked that they sum up to 64.

**Weight multiplicities**

The weights of the character are available (with their coefficients) through the method `weight_multiplicities`. Continuing from the example in the last section:

```
sage: pprint(chi.weight_multiplicities())
{(0, 0, 0): 8, (-1, 0, 0): 4, (-1, -1, 0): 2, (-1, -1, -1): 1,
 (-1, -1, 1): 1, (-1, 1, 0): 2, (-1, 1, -1): 1, (-1, 1, 1): 1,
 (-1, 0, -1): 2, (-1, 0, 1): 2, (1, 0, 0): 4, (1, -1, 0): 2,
 (1, -1, -1): 1, (1, -1, 1): 1, (1, 1, 0): 2, (1, 1, -1): 1,
 (1, 1, 1): 1, (1, 0, -1): 2, (1, 0, 1): 2, (0, -1, 0): 4,
 (0, -1, -1): 2, (0, -1, 1): 2, (0, 1, 0): 4, (0, 1, -1): 2,
 (0, 1, 1): 2, (0, 0, -1): 4, (0, 0, 1): 4}
```

Each key of this dictionary is a weight, and its value is the multiplicity of that weight in the character.

**Example**

Suppose that we wish to compute the integral

\[
\int_{U(n)} |\text{tr}(g)|^{2k} \, dg
\]

for various \(n\). Here \(U(n)\) is the unitary group, which is the maximal compact subgroup of \(GL(n, \mathbb{C})\). The irreducible unitary representations of \(U(n)\) may be regarded as the basis elements of the WeylCharacterRing of type \(A_r\), where \(r = n - 1\) so we might work in that ring. The trace \(\text{tr}(g)\) is then just the character of the standard representation. We may realize it in the WeylCharacterRing by taking the first fundamental weight and coercing it into the ring. For example, if \(k = 5\) and \(n = 3\) so \(r = 2\):

```
sage: A2 = WeylCharacterRing("A2")
sage: fw = A2.fundamental_weights(); fw
Finite family {1: (1, 0, 0), 2: (1, 1, 0)}
sage: tr = A2(fw[1]); tr
A2(1,0,0)
```

We may compute the norm square the character \(\text{tr}^5\) by decomposing it into irreducibles, and taking the sum of the squares of their multiplicities. By Schur orthogonality, this gives the inner product of the \(\text{tr}(g)^5\) with itself, that is, the integral of \(|\text{tr}(g)|^{10}\):

```
sage: sum(d^2 for d in (tr^5).coefficients())
103
```

So far we have been working with \(n = 3\). For general \(n\):

```
sage: def f(n,k):
......    R = WeylCharacterRing(['A',n-1])
......    tr = R(R.fundamental_weights()[1])
......    return sum(d^2 for d in (tr^k).coefficients())
sage: [f(n,5) for n in [2..7]]
[42, 103, 119, 120, 120, 120]
```
We see that the 10-th moment of $tr(g)$ is just $5!$ when $n$ is sufficiently large. What if we fix $n$ and vary $k$?

\[
sage: \text{[f(2,k) for } k \text{ in [1..10])]}
n\text{[1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796]}
sage: \text{[catalan_number(k) for } k \text{ in [1..10]]}
n\text{[1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796]}
\]

**Frobenius-Schur indicator**

The Frobenius-Schur indicator of an irreducible representation of a compact Lie group $G$ with character $\chi$ is:

\[
\int_G \chi(g^2) \, dg
\]

The Haar measure is normalized so that $\text{vol}(G) = 1$. The Frobenius-Schur indicator equals 1 if the representation is real (orthogonal), $-1$ if it is quaternionic (symplectic) and 0 if it is complex (not self-contragredient). This is a method of weight ring elements corresponding to irreducible representations. Let us compute the Frobenius-Schur indicators of the spin representations of some odd spin groups:

\[
sage: \text{def spinrepn(r):}
....: \quad R = WeylCharacterRing(['B',r])
....: \quad return R(R.fundamental_weights()[r])
....:
sage: spinrepn(3)
B3(1/2,1/2,1/2)
sage: \text{for } r \text{ in [1..4]:}
....: \quad print("{} \{}").format(r, spinrepn(r).frobenius_schur_indicator())
1 -1
2 -1
3 1
4 1
\]

Here we have defined a function that returns the spin representation of the group $\text{spin}(2r + 1)$ with Cartan type $['B',r]$, then computed the Frobenius-Schur indicators for a few values. From this experiment we see that the spin representations of $\text{spin}(3)$ and $\text{spin}(5)$ are symplectic, while those of $\text{spin}(7)$ and $\text{spin}(9)$ are orthogonal.

**Symmetric and exterior powers**

Sage can compute symmetric and exterior powers of a representation:

\[
sage: B3 = WeylCharacterRing("B3", style="coroots")
sage: spin = B3(0,0,1); spin.degree()
8
sage: spin.exterior_power(2)
B3(1,0,0) + B3(0,1,0)
sage: spin.exterior_square()
B3(1,0,0) + B3(0,1,0)
sage: spin.exterior_power(5)
B3(0,0,1) + B3(1,0,1)
sage: spin.symmetric_power(5)
B3(0,0,1) + B3(0,0,3) + B3(0,0,5)
\]
The $k$-th exterior square of a representation is zero if $k$ is greater than the degree of the representation. However the $k$-th symmetric power is nonzero for all $k$.

The tensor square of any representation decomposes as the direct sum of the symmetric and exterior squares:

```python
sage: C4 = WeylCharacterRing("C4",style="coroots")
sage: chi = C4(1,0,0,0); chi.degree()
8
sage: chi.symmetric_square()
C4(2,0,0,0)
sage: chi.exterior_square()
C4(0,0,0,0) + C4(0,1,0,0)
sage: chi^2 == chi.symmetric_square() + chi.exterior_square()
True
```

Since in this example the exterior square contains the trivial representation we expect the Frobenius-Schur indicator to be $-1$, and indeed it is:

```python
sage: chi = C4(1,0,0,0)
sage: chi.frobenius_schur_indicator()
-1
```

This is not surprising since this is the standard representation of a symplectic group, which is symplectic by definition!

**Weyl dimension formula**

If the representation is truly large you will not be able to construct it in the Weyl character ring, since internally it is represented by a dictionary of its weights. If you want to know its degree, you can still compute that since Sage implements the Weyl dimension formula. The degree of the representation is implemented as a method of its highest weight vector:

```python
sage: L = RootSystem("E8").ambient_space()
sage: [L.weyl_dimension(f) for f in L.fundamental_weights()]
[3875, 147250, 6696000, 6899079264, 146325270, 2450240, 30380, 248]
```

It is a fact that for any compact Lie group if $\rho$ is the Weyl vector (half the sum of the positive roots) then the degree of the irreducible representation with highest weight $\rho$ equals $2^N$ where $N$ is the number of positive roots. Let us check this for $E_8$. In this case $N = 120$:

```python
sage: L = RootSystem("E8").ambient_space()
sage: len(L.positive_roots())
120
sage: 2^120
1329227995784915872903807060280344576
sage: L.weyl_dimension(L.rho())
1329227995784915872903807060280344576
```
**SL versus GL**

Sage takes the weight space for type ['A', r] to be \( r + 1 \) dimensional. As a by-product, if you create the Weyl character ring with the command:

```python
sage: A2 = WeylCharacterRing("A2")
```

Then you are effectively working with \( GL(3) \) instead of \( SL(3) \). For example, the determinant is the character \( A2(1, 1, 1) \). However, as we will explain later, you can work with \( SL(3) \) if you like, so long as you are willing to work with fractional weights. On the other hand if you create the Weyl character ring with the command:

```python
sage: A2 = WeylCharacterRing("A2", style="coroots")
```

Then you are working with \( SL(3) \).

There are some advantages to this arrangement:

- The group \( GL(r + 1) \) arises frequently in practice. For example, even if you care mainly about semisimple groups, the group \( GL(r + 1) \) may arise as a Levi subgroup.
- It avoids fractional weights. If you want to work with \( SL(3) \) the fundamental weights are \((2/3, -1/3, -1/3)\) and \((1/3, 1/3, -2/3)\). If you work instead with \( GL(3) \), they are \((1, 0, 0)\) and \((1, 1, 0)\). For many mathematical purposes it doesn’t make any difference which you use. This is because the difference between \((2/3, -1/3, -1/3)\) and \((1, 0, 0)\) is a vector that is orthogonal to all the simple roots. Thus these vectors are interchangeable. But for convenience avoiding fractional weights is advantageous.

However if you want to be an \( SL \) purist, Sage will support you. The weight space for \( SL(3) \) can be taken to be the hyperplane in \( \mathbb{Q}^3 \) consisting of vectors \((a, b, c)\) with \(a + b + c = 0\). The fundamental weights for \( SL(3) \) are then \((2/3, -1/3, -1/3)\) and \((1/3, 1/3, -2/3)\), though Sage will tell you they are \((1, 0, 0)\) and \((1, 1, 0)\). The work-around is to filter them through the method `coerce_to_sl` as follows:

```python
sage: A2 = WeylCharacterRing("A2")
sage: [fw1,fw2] = [w.coerce_to_sl() for w in A2.fundamental_weights()]
sage: [standard, contragredient] = [A2(fw1), A2(fw2)]
sage: standard, contragredient
(A2(2/3,-1/3,-1/3), A2(1/3,1/3,-2/3))
sage: standard^contragredient
A2(0,0,0) + A2(1,0,-1)
```

Sage is not confused by the fractional weights. Note that if you use coroot notation, you are working with \( SL \) automatically:

```python
sage: A2 = WeylCharacterRing("A2", style="coroots")
sage: A2(1,0).weight_multiplicities()
```

There is no convenient way to create the determinant in the Weyl character ring if you adopt the coroot style.

Just as we coerced the fundamental weights into the \( SL \) weight lattice, you may need to coerce the Weyl vector \( \rho \) if you are working with \( SL \). The default value for \( \rho \) in type \( A_r \) is \((r, r - 1, \ldots, 0)\), but if you are an \( SL \) purist you want

\[
\left( \frac{r}{2}, \frac{r}{2} - 1, \ldots, \frac{r}{2} \right).
\]

Therefore take the value of \( \rho \) that you get from the method of the ambient space and coerce it into \( SL \):
sage: A2 = WeylCharacterRing("A2", style="coroots")
sage: rho = A2.space().rho().coerce_to_sl(); rho
(1, 0, -1)
sage: rho == (1/2)*sum(A2.space().positive_roots())
True

You do not need to do this for other Cartan types. If you are working with (say) $F_4$ then a $\rho$ is a $\rho$:  

sage: F4 = WeylCharacterRing("F4")
sage: L = F4.space()
sage: rho = L.rho()
sage: rho == (1/2)*sum(L.positive_roots())
True

Integration

Suppose that we wish to compute the integral  
$$\int_{U(n)} |tr(g)|^{2k} dg$$  
for various $n$. Here $U(n)$ is the unitary group, which is the maximal compact subgroup of $GL(n, \mathbb{C})$, and $dg$ is the Haar measure on $U(n)$, normalized so that the volume of the group is 1.

The irreducible unitary representations of $U(n)$ may be regarded as the basis elements of the WeylCharacterRing of type $A_r$, where $r = n - 1$ so we might work in that ring. The trace $tr(g)$ is then just the character of the standard representation. We may realize it in the WeylCharacterRing by taking the first fundamental weight and coercing it into the ring. For example, if $k = 5$ and $n = 3$ so $r = 2$:

sage: A2 = WeylCharacterRing("A2")
sage: fw = A2.fundamental_weights(); fw
Finite family {1: (1, 0, 0), 2: (1, 1, 0)}
sage: tr = A2(fw[1]); tr
A2(1,0,0)

We may compute the norm square the character $tr^5$ by decomposing it into irreducibles, and taking the sum of the squares of their multiplicities. By Schur orthogonality, this gives the inner product of the $tr(g)^5$ with itself, that is, the integral of $|tr(g)|^{10}$:

sage: tr^5
5*A2(2,2,1) + 6*A2(3,1,1) + 5*A2(3,2,0) + 4*A2(4,1,0) + A2(5,0,0)
sage: sorted((tr^5).coefficients())
[1, 4, 5, 5, 6]
sage: sum(x^2 for x in (tr^5).coefficients())
103

So far we have been working with $n = 3$. For general $n$:

sage: def f(n,k):
....:     R = WeylCharacterRing(["A",n-1])
....:     tr = R(R.fundamental_weights()[1])
(continues on next page)
We see that the 10-th moment of \( tr(g) \) is just \( 5! \) when \( n \) is sufficiently large. What if we fix \( n \) and vary \( k \)?

\[
\text{sage: } [f(2,k) \text{ for } k \text{ in } [1..10]]
\]
\[1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796\]

\[
\text{sage: } [\text{catalan_number}(k) \text{ for } k \text{ in } [1..10]]
\]
\[1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796\]

Invariants and multiplicities

Sometimes we are only interested in the multiplicity of the trivial representation in some character. This may be found by the method `invariant_degree`. Continuing from the preceding example,

\[
\text{sage: } \text{A2} = \text{WeylCharacterRing("A2",style="coroots")}
\]
\[
\text{sage: } \text{ad} = \text{A2}(1,1)
\]
\[
\text{sage: } [\text{ad.symmetric_power(k).invariant_degree()} \text{ for } k \text{ in } [0..6]]
\]
\[1, 0, 1, 1, 1, 1, 2\]
\[
\text{sage: } [\text{ad.exterior_power(k).invariant_degree()} \text{ for } k \text{ in } [0..6]]
\]
\[1, 0, 0, 1, 0, 1, 0\]

If we want the multiplicity of some other representation, we may obtain that using the method `multiplicity`:

\[
\text{sage: } (\text{ad}^3).\text{multiplicity}(\text{ad})
\]
\[8\]

Weight Rings

You may wish to work directly with the weights of a representation.

Weyl character ring elements are represented internally by a dictionary of their weights with multiplicities. However these are subject to a constraint: the coefficients must be invariant under the action of the Weyl group.

The `WeightRing` is also a ring whose elements are represented internally by a dictionary of their weights with multiplicities, but it is not subject to this constraint of Weyl group invariance. The weights are allowed to be fractional, that is, elements of the ambient space. In other words, the weight ring is the group algebra over the ambient space of the weight lattice.

To create a `WeightRing` first construct the `WeylCharacterRing`, then create the `WeightRing` as follows:

\[
\text{sage: } \text{A2} = \text{WeylCharacterRing(['A',2])}
\]
\[
\text{sage: } \text{a2} = \text{WeightRing(A2)}
\]

You may coerce elements of the `WeylCharacterRing` into the weight ring. For example, if you want to see the weights of the adjoint representation of \( GL(3) \), you may use the method `mlist`, but another way is to coerce it into the weight ring:
sage: from pprint import pprint
sage: A2 = WeylCharacterRing(['A',2])

sage: ad = A2(1,0,-1)
sage: pprint(ad.weight_multiplicities())
{(0, 0, 0): 2, (-1, 1, 0): 1, (-1, 0, 1): 1, (1, -1, 0): 1, (1, 0, -1): 1, (0, -1, 1): 1, (0, 1, -1): 1}

This command produces a dictionary of the weights that appear in the representation, together with their multiplicities. But another way of getting the same information, with an aim of working with it, is to coerce it into the weight ring:

sage: a2 = WeightRing(A2)
sage: a2(ad)
2*a2(0,0,0) + a2(-1,1,0) + a2(-1,0,1) + a2(1,-1,0) + a2(1,0,-1) + a2(0,-1,1) + a2(0,1,-1)

For example, the Weyl denominator formula is usually written this way:

$$\prod_{\alpha \in \Phi^+} \left( e^{\alpha/2} - e^{-\alpha/2} \right) = \sum_{w \in W} (-1)^{l(w)} e^{w(\rho)}.$$ 

The notation is as follows. Here if $\lambda$ is a weight, or more generally, an element of the ambient space, then $e^\lambda$ means the image of $\lambda$ in the group algebra of the ambient space of the weight lattice $\lambda$. Since this group algebra is just the weight ring, we can interpret $e^\lambda$ as its image in the weight ring.

Let us confirm the Weyl denominator formula for $A_2$:

sage: A2 = WeylCharacterRing("A2")
sage: a2 = WeightRing(A2)
sage: L = A2.space()
sage: W = L.weyl_group()
sage: rho = L.rho().coerce_to_sl()

sage: lhs = prod(a2(alpha/2)-a2(-alpha/2) for alpha in L.positive_roots()); lhs
a2(-1,1,0) - a2(-1,0,1) - a2(1,-1,0) + a2(1,0,-1) + a2(0,-1,1) - a2(0,1,-1)

sage: rhs = sum((-1)^(w.length())*a2(w.action(rho)) for w in W); rhs
a2(-1,1,0) - a2(-1,0,1) - a2(1,-1,0) + a2(1,0,-1) + a2(0,-1,1) - a2(0,1,-1)

sage: lhs == rhs
True

Note that we have to be careful to use the right value of $\rho$. The reason for this is explained in SL versus GL.

We have seen that elements of the WeylCharacterRing can be coerced into the WeightRing. Elements of the WeightRing can be coerced into the WeylCharacterRing provided they are invariant under the Weyl group.

Maximal Subgroups and Branching Rules

Branching rules

If $G$ is a Lie group and $H$ is a subgroup, one often needs to know how representations of $G$ restrict to $H$. Irreducibles usually do not restrict to irreducibles. In some cases the restriction is regular and predictable, in other cases it is chaotic. In some cases it follows a rule that can be described combinatorially, but the combinatorial description is subtle. The description of how irreducibles decompose into irreducibles is called a branching rule.

References for this topic:

- [FauserEtAl2006]
- [King1975]
Sage can compute how a character of $G$ restricts to $H$. It does so not by memorizing a combinatorial rule, but by computing the character and restricting the character to a maximal torus of $H$. What Sage has memorized (in a series of built-in encoded rules) are the various embeddings of maximal tori of maximal subgroups of $G$. The maximal subgroups of Lie groups were determined in [Dynkin1952]. This approach to computing branching rules has a limitation: the character must fit into memory and be computable by Sage’s internal code in real time.

It is sufficient to consider the case where $H$ is a maximal subgroup of $G$, since if this is known then one may branch down successively through a series of subgroups, each maximal in its predecessors. A problem is therefore to understand the maximal subgroups in a Lie group, and to give branching rules for each, and a goal of this tutorial is to explain the embeddings of maximal subgroups.

Sage has a class BranchingRule for branching rules. The function branching_rule returns elements of this class. For example, the natural embedding of $Sp(4)$ into $SL(4)$ corresponds to the branching rule that we may create as follows:

```sage
sage: b = branching_rule("A3","C2",rule="symmetric"); b
symmetric branching rule A3 => C2
```

The name “symmetric” of this branching rule will be explained further later, but it means that $Sp(4)$ is the fixed subgroup of an involution of $SL(4)$. Here $A3$ and $C2$ are the Cartan types of the groups $G = SL(4)$ and $H = Sp(4)$.

Now we may see how representations of $SL(4)$ decompose into irreducibles when they are restricted to $Sp(4)$:

```sage
sage: A3 = WeylCharacterRing("A3",style="coroots")
sage: chi = A3(1,0,1); chi.degree()
15
sage: C2 = WeylCharacterRing("C2",style="coroots")
sage: chi.branch(C2,rule=b)
C2(0,1) + C2(2,0)
```

Alternatively, we may pass `chi` to `b` as an argument of its branch method, which gives the same result:

```sage
sage: b.branch(chi)
C2(0,1) + C2(2,0)
```

It is believed that the built-in branching rules of Sage are sufficient to handle all maximal subgroups and this is certainly the case when the rank if less than or equal to 8.

However, if you want to branch to a subgroup that is not maximal you may not find a built-in branching rule. We may compose branching rules to build up embeddings. For example, here are two different embeddings of $Sp(4)$ with Cartan type $C2$ in $Sp(8)$, with Cartan type $C4$. One embedding factors through $Sp(4) \times Sp(4)$, while the other factors through $SL(4)$. To check that the embeddings are not conjugate, we branch a (randomly chosen) representation. Observe that we do not have to build the intermediate Weyl character rings.

```sage
sage: C4 = WeylCharacterRing("C4",style="coroots")
sage: b1 = branching_rule("C4","A3","levi")*branching_rule("A3","C2","symmetric"); b1
composite branching rule C4 => (levi) A3 => (symmetric) C2
sage: b2 = branching_rule("C4","C2xC2","orthogonal_sum")*branching_rule("C2xC2","C2", "proj1"); b2
composite branching rule C4 => (orthogonal_sum) C2xC2 => (proj1) C2
sage: C2 = WeylCharacterRing("C2",style="coroots")
sage: C4 = WeylCharacterRing("C4",style="coroots")
sage: [C4(2,0,0,1).branch(C2, rule=br) for br in [b1,b2]]
```

(continues on next page)
What's in a branching rule?

The essence of the branching rule is a function from the weight lattice of $G$ to the weight lattice of the subgroup $H$, usually implemented as a function on the ambient vector spaces. Indeed, we may conjugate the embedding so that a Cartan subalgebra $U$ of $H$ is contained in a Cartan subalgebra $T$ of $G$. Since the ambient vector space of the weight lattice of $G$ is Lie$(T)^*$, we get map Lie$(T)^* \rightarrow$ Lie$(U)^*$, and this must be implemented as a function. For speed, the function usually just returns a list, which can be coerced into Lie$(U)^*$.

```
sage: b = branching_rule("A3","C2","symmetric")
sage: for r in RootSystem("A3").ambient_space().simple_roots():
    ....:     print("{} {}\n".format(r, b(r)))
(1, -1, 0, 0) [1, -1]
(0, 1, -1, 0) [0, 2]
(0, 0, 1, -1) [1, -1]
```

We could conjugate this map by an element of the Weyl group of $G$, and the resulting map would give the same decomposition of representations of $G$ into irreducibles of $H$. However it is a good idea to choose the map so that it takes dominant weights to dominant weights, and, insofar as possible, simple roots of $G$ into simple roots of $H$. This includes sometimes the affine root $\alpha_0$ of $G$, which we recall is the negative of the highest root.

The branching rule has a `describe()` method that shows how the roots (including the affine root) restrict. This is a useful way of understanding the embedding. You might want to try it with various branching rules of different kinds, "extended", "symmetric", "levi" etc.

```
sage: b.describe()
```

The extended Dynkin diagram of $G$ and the ordinary Dynkin diagram of $H$ are shown for reference, and 3 $\Rightarrow$ 1 means that the third simple root $\alpha_3$ of $G$ restricts to the first simple root of $H$. In this example, the affine root does not restrict
to a simple roots, so it is omitted from the list of restrictions. If you add the parameter verbose=True you will be shown the restriction of all simple roots and the affine root, and also the restrictions of the fundamental weights (in coroot notation).

**Maximal subgroups**

Sage has a database of maximal subgroups for every simple Cartan type of rank $\leq 8$. You may access this with the `maximal_subgroups` method of the Weyl character ring:

```
sage: E7 = WeylCharacterRing("E7",style="coroots")
sage: E7.maximal_subgroups()
A7:branching_rule("E7","A7","extended")
E6:branching_rule("E7","E6","levi")
A2:branching_rule("E7","A2","miscellaneous")
A1:branching_rule("E7","A1","iii")
A1:branching_rule("E7","A1","iv")
A1xF4:branching_rule("E7","A1xF4","miscellaneous")
G2xC3:branching_rule("E7","G2xC3","miscellaneous")
A1xG2:branching_rule("E7","A1xG2","miscellaneous")
A1xA1:branching_rule("E7","A1xA1","miscellaneous")
A1xD6:branching_rule("E7","A1xD6","extended")
A5xA2:branching_rule("E7","A5xA2","extended")
```

It should be understood that there are other ways of embedding $\mathfrak{a}_2 = \text{SL}(3)$ into the Lie group $E_7$, but only one way as a maximal subgroup. On the other hand, there are but only one way to embed it as a maximal subgroup. The embedding will be explained below. You may obtain the branching rule as follows, and use it to determine the decomposition of irreducible representations of $E_7$ as follows:

```
sage: b = E7.maximal_subgroup("A2"); b
miscellaneous branching rule E7 => A2
sage: E7, A2 = [WeylCharacterRing(x,style="coroots") for x in ["E7","A2"]]
sage: E7(1,0,0,0,0,0,0).branch(A2,rule=b)
A2(1,1) + A2(4,4)
```

This gives the same branching rule as just pasting line beginning to the right of the colon onto the command line:

```
sage: branching_rule("E7","A2","miscellaneous")
miscellaneous branching rule E7 => A2
```

There are two distinct embeddings of $\mathfrak{a}_1 = \text{SL}(2)$ into $E_7$ as maximal subgroups, so the `maximal_subgroup` method will return a list of rules:

```
sage: WeylCharacterRing("E7").maximal_subgroup("A1")
[iii branching rule E7 => A1, iv branching rule E7 => A1]
```

The list of maximal subgroups returned by the `maximal_subgroups` method for irreducible Cartan types of rank up to 8 is believed to be complete up to outer automorphisms. You may want a list that is complete up to inner automorphisms. For example, $E_6$ has a nontrivial Dynkin diagram automorphism so it has an outer automorphism that is not inner:

```
sage: E6, A2xG2 = [WeylCharacterRing(x,style="coroots") for x in ["E6","A2xG2"]]
sage: b = E6.maximal_subgroup("A2xG2"); b
miscellaneous branching rule E6 => A2xG2
sage: E6(1,0,0,0,0,0).branch(A2xG2,rule=b)
```

(continues on next page)
A\times G_2(0,1,1,0) + A\times G_2(2,0,0,0)
sage: E6(0,0,0,0,0,1).branch(A\times G_2, rule=b)
A\times G_2(1,0,1,0) + A\times G_2(0,2,0,0)

Since as we see the two 27 dimensional irreducibles (which are interchanged by the outer automorphism) have different branching, the $A_2 \times G_2$ subgroup is changed to a different one by the outer automorphism. To obtain the second branching rule, we compose the given one with this automorphism:
sage: b1 = branching_rule("E6","E6","automorphic")*b; b1
composite branching rule E6 => (automorphic) E6 => (miscellaneous) A\times G_2

Levi subgroups

A Levi subgroup may or may not be maximal. They are easily classified. If one starts with a Dynkin diagram for $G$ and removes a single node, one obtains a smaller Dynkin diagram, which is the Dynkin diagram of a smaller subgroup $H$.

For example, here is the $A_3$ Dynkin diagram:
sage: A3 = WeylCharacterRing("A3")
sage: A3.dynkin_diagram()
O---O---O
1 2 3
A3

We see that we may remove the node 3 and obtain $A_2$, or the node 2 and obtain $A_1 \times A_1$. These correspond to the Levi subgroups $GL(3)$ and $GL(2) \times GL(2)$ of $GL(4)$.

Let us construct the irreducible representations of $GL(4)$ and branch them down to these down to $GL(3)$ and $GL(2) \times GL(2)$:
sage: reps = [A3(v) for v in A3.fundamental_weights()]; reps
[A3(1,0,0,0), A3(1,1,0,0), A3(1,1,1,0)]
sage: A2 = WeylCharacterRing("A2")
sage: A1xA1 = WeylCharacterRing("A1xA1")
sage: [pi.branch(A2, rule="levi") for pi in reps]
[A2(0,0,0) + A2(1,0,0), A2(1,0,0) + A2(1,1,0), A2(1,1,0) + A2(1,1,1),
A1xA1(1,0,0,0) + A1xA1(0,0,1,0) + A1xA1(0,0,1,1),
A1xA1(1,1,1,0) + A1xA1(1,0,1,1)]

Let us redo this calculation in coroot notation. As we have explained, coroot notation does not distinguish between representations of $GL(4)$ that have the same restriction to $SL(4)$, so in effect we are now working with the groups $SL(4)$ and its Levi subgroups $SL(3)$ and $SL(2) \times SL(2)$, which is the derived group of its Levi subgroup:
sage: A3 = WeylCharacterRing("A3", style="coroots")
sage: reps = [A3(v) for v in A3.fundamental_weights()]; reps
[A3(1,0,0,0), A3(1,1,0,0), A3(1,1,1,0)]
sage: A2 = WeylCharacterRing("A2", style="coroots")
sage: A1xA1 = WeylCharacterRing("A1xA1", style="coroots")
sage: [pi.branch(A2, rule="levi") for pi in reps]
Now we may observe a distinction difference in branching from

\[ GL(4) \to GL(2) \times GL(2) \]

versus

\[ SL(4) \to SL(2) \times SL(2). \]

Consider the representation \( A_3(0,1,0) \), which is the six dimensional exterior square. In the coroot notation, the restriction contained two copies of the trivial representation, \( 2^a A_{1 \times A_1}(\phi, \phi) \). The other way, we had instead three distinct representations in the restriction, namely \( A_{1 \times A_1}(1,1,0,0) \) and \( A_{1 \times A_1}(0,0,1,1) \), that is, \( \text{det} \otimes 1 \) and \( 1 \otimes \text{det} \).

The Levi subgroup \( A_{1 \times A_1} \) is actually not maximal. Indeed, we may factor the embedding:

\[ SL(2) \times SL(2) \to Sp(4) \to SL(4). \]

Therefore there are branching rules \( A_3 \to C_2 \) and \( C_2 \to A_2 \), and we could accomplish the branching in two steps, thus:

```python
sage: A3 = WeylCharacterRing("A3", style="coroots")
sage: C2 = WeylCharacterRing("C2", style="coroots")
sage: B2 = WeylCharacterRing("B2", style="coroots")
sage: D2 = WeylCharacterRing("D2", style="coroots")
sage: A1xA1 = WeylCharacterRing("A1xA1", style="coroots")
sage: reps = [A3(fw) for fw in A3.fundamental_weights()]
```

As you can see, we’ve redone the branching rather circuitously this way, making use of the branching rules \( A_3 \to C_2 \) and \( C_2 \to D_2 \), and two accidental isomorphisms \( C_2 = B_2 \) and \( D_2 = A_1 \times A_1 \). It is much easier to go in one step using rule="levi", but reassuring that we get the same answer!

### Subgroups classified by the extended Dynkin diagram

It is also true that if we remove one node from the extended Dynkin diagram that we obtain the Dynkin diagram of a subgroup. For example:

```python
sage: G2 = WeylCharacterRing("G2", style="coroots")
sage: G2.extended_dynkin_diagram()
```

Observe that by removing the 1 node that we obtain an \( A_2 \) Dynkin diagram. Therefore the exceptional group \( G_2 \) contains a copy of \( SL(3) \). We branch the two representations of \( G_2 \) corresponding to the fundamental weights to this copy of \( A_2 \):
The two representations of $G_2$, of degrees 7 and 14 respectively, are the action on the octonions of trace zero and the adjoint representation.

For embeddings of this type, the rank of the subgroup $H$ is the same as the rank of $G$. This is in contrast with embeddings of Levi type, where $H$ has rank one less than $G$.

**Levi subgroups of $G_2$**

The exceptional group $G_2$ has two Levi subgroups of type $A_1$. Neither is maximal, as we can see from the extended Dynkin diagram: the subgroups $A_1 \times A_1$ and $A_2$ are maximal and each contains a Levi subgroup. (Actually $A_1 \times A_1$ contains a conjugate of both.) Only the Levi subgroup containing the short root is implemented as an instance of rule="levi". To obtain the other, use the rule:

```python
sage: branching_rule("G2","A2","extended")*branching_rule("A2","A1","levi")
```

which branches to the $A_1$ Levi subgroup containing a long root.

**Orthogonal and symplectic subgroups of orthogonal and symplectic groups**

If $G = SO(n)$ then $G$ has a subgroup $SO(n - 1)$. Depending on whether $n$ is even or odd, we thus have branching rules ['D',r] to ['B',r-1] or ['B',r] to ['D',r]. These are handled as follows:

```python
sage: branching_rule("B4","D4",rule="extended")
extended branching rule B4 => D4
sage: branching_rule("D4","B3",rule="symmetric")
symmetric branching rule D4 => B3
```

If $G = SO(r + s)$ then $G$ has a subgroup $SO(r) \times SO(s)$. This lifts to an embedding of the universal covering groups

$$\text{spin}(r) \times \text{spin}(s) \to \text{spin}(r + s).$$

Sometimes this embedding is of extended type, and sometimes it is not. It is of extended type unless $r$ and $s$ are both odd. If it is of extended type then you may use rule="extended". In any case you may use rule="orthogonal_sum". The name refer to the origin of the embedding $SO(r) \times SO(s) \to SO(r + s)$ from the decomposition of the underlying quadratic space as a direct sum of two orthogonal subspaces.

There are four cases depending on the parity of $r$ and $s$. For example, if $r = 2k$ and $s = 2l$ we have an embedding:

```python
['D',k] x ['D',l] --> ['D',k+l]
```

This is of extended type. Thus consider the embedding $D4 \times D3 \to D7$. Here is the extended Dynkin diagram:
Removing the 4 vertex results in a disconnected Dynkin diagram:

\[
\begin{array}{cc}
\text{\ O \ O \ 7} \\
| \\
| \\
\text{\ O---O---O}\ \\
\end{array}
\]

This is $D_4 \times D_3$. Therefore use the “extended” branching rule:

```
sage: D7 = WeylCharacterRing("D7", style="coroots")
sage: D4xD3 = WeylCharacterRing("D4xD3", style="coroots")
sage: spin = D7(D7.fundamental_weights()[7]); spin
D7(0,0,0,0,0,0,1)
sage: spin.branch(D4xD3, rule="extended")
D4xD3(0,0,1,0,0,1,0) + D4xD3(0,0,0,1,0,0,1)
```

But we could equally well use the “orthogonal_sum” rule:

```
sage: spin.branch(D4xD3, rule="orthogonal_sum")
D4xD3(0,0,1,0,0,1,0) + D4xD3(0,0,0,1,0,0,1)
```

Similarly we have embeddings:

\[
[D',k] \times [B',l] \rightarrow [B',k+l]
\]

These are also of extended type. For example consider the embedding of $D_3 \times B_2 \rightarrow B_5$. Here is the $B_5$ extended Dynkin diagram:

\[
\begin{array}{cc}
\text{\ O\ O} \\
| \\
| \\
\text{\ O---O---O---O=>=O} \\
\end{array}
\]

Removing the 3 node gives:

\[
\begin{array}{cc}
\text{\ O\ O} \\
| \\
| \\
\text{\ O---O---O---O=>=O} \\
\end{array}
\]

and this is the Dynkin diagram or $D_3 \times B_2$. For such branchings we again use either rule="extended" or rule="orthogonal_sum".

Finally, there is an embedding

\[
[B',k] \times [B',l] \rightarrow [D',k+l+1]
\]

This is not of extended type, so you may not use rule="extended". You must use rule="orthogonal_sum":
Non-maximal Levi subgroups and Projection from Reducible Types

Not all Levi subgroups are maximal. Recall that the Dynkin-diagram of a Levi subgroup $H$ of $G$ is obtained by removing a node from the Dynkin diagram of $G$. Removing the same node from the extended Dynkin diagram of $G$ results in the Dynkin diagram of a subgroup of $G$ that is strictly larger than $H$. However this subgroup may or may not be proper, so the Levi subgroup may or may not be maximal.

If the Levi subgroup is not maximal, the branching rule may or may not be implemented in Sage. However if it is not implemented, it may be constructed as a composition of two branching rules.

For example, prior to Sage-6.1 `branching_rule("E6","A5","levi")` returned a not-implemented error and the advice to branch to $A5xA1$. And we can see from the extended Dynkin diagram of $E_6$ that indeed $A_5$ is not a maximal subgroup, since removing node 2 from the extended Dynkin diagram (see below) gives $A5xA1$. To construct the branching rule to $A_5$ we may proceed as follows:

```
sage: b = branching_rule("E6","A5xA1","extended")*branching_rule("A5xA1","A5","proj1"); b
composite branching rule E6 => (extended) A5xA1 => (proj1) A5
sage: E6 = WeylCharacterRing("E6",style="coroots")
sage: A5 = WeylCharacterRing("A5",style="coroots")
sage: E6(0,1,0,0,0,0).branch(A5,rule=b)
3*A5(0,0,0,0,0) + 2*A5(0,0,1,0,0) + A5(1,0,0,1,0)
sage: b.describe()
O 0
   |
   |
O 2
   |
   |
O---O---O---O---O
1 3 4 5 6

E6-
root restrictions E6 => A5:

O---O---O---O---O
1 2 3 4 5
A5
0 => (zero)
1 => 1
3 => 2
4 => 3
5 => 4
6 => 5
```
For more detailed information use verbose=True

Note that it is not necessary to construct the Weyl character ring for the intermediate group $A_5 \times A_1$.

This last example illustrates another common problem: how to extract one component from a reducible root system. We used the rule "proj1" to extract the first component. We could similarly use "proj2" to get the second, or more generally any combination of components:

```
sage: branching_rule("A2xB2xG2","A2xG2","proj13")
```

proj13 branching rule A2xB2xG2 => A2xG2

**Symmetric subgroups**

If $G$ admits an outer automorphism (usually of order two) then we may try to find the branching rule to the fixed subgroup $H$. It can be arranged that this automorphism maps the maximal torus $T$ to itself and that a maximal torus $U$ of $H$ is contained in $T$.

Suppose that the Dynkin diagram of $G$ admits an automorphism. Then $G$ itself admits an outer automorphism. The Dynkin diagram of the group $H$ of invariants may be obtained by "folding" the Dynkin diagram of $G$ along the automorphism. The exception is the branching rule $GL(2r) \to SO(2r)$.

Here are the branching rules that can be obtained using rule="symmetric".

<table>
<thead>
<tr>
<th>$G$</th>
<th>$H$</th>
<th>Cartan Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>$GL(2r)$</td>
<td>$Sp(2r)$</td>
<td>$['A',2r-1] \Rightarrow ['C',r]$</td>
</tr>
<tr>
<td>$GL(2r+1)$</td>
<td>$SO(2r+1)$</td>
<td>$['A',2r] \Rightarrow ['B',r]$</td>
</tr>
<tr>
<td>$GL(2r)$</td>
<td>$SO(2r)$</td>
<td>$['A',2r-1] \Rightarrow ['D',r]$</td>
</tr>
<tr>
<td>$SO(2r)$</td>
<td>$SO(2r-1)$</td>
<td>$['D',r] \Rightarrow ['B',r-1]$</td>
</tr>
<tr>
<td>$E_6$</td>
<td>$F_4$</td>
<td>$['E',6] \Rightarrow ['F',4]$</td>
</tr>
</tbody>
</table>

**Tensor products**

If $G_1$ and $G_2$ are Lie groups, and we have representations $\pi_1 : G_1 \to GL(n)$ and $\pi_2 : G_2 \to GL(m)$ then the tensor product is a representation of $G_1 \times G_2$. It has its image in $GL(nm)$ but sometimes this is conjugate to a subgroup of $SO(nm)$ or $Sp(nm)$. In particular we have the following cases.

<table>
<thead>
<tr>
<th>Group</th>
<th>Subgroup</th>
<th>Cartan Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>$GL(rs)$</td>
<td>$GL(r) \times GL(s)$</td>
<td>$['A',rs-1] \Rightarrow ['A',r-1] \times ['A',s-1]$</td>
</tr>
<tr>
<td>$SO(4rs+2r+2s+1)$</td>
<td>$SO(2r+1) \times SO(2s+1)$</td>
<td>$['B',2rs+r+s] \Rightarrow ['B',r] \times ['B',s]$</td>
</tr>
<tr>
<td>$SO(4rs+2s)$</td>
<td>$SO(2r) \times SO(2s)$</td>
<td>$['D',2rs+s] \Rightarrow ['B',r] \times ['D',s]$</td>
</tr>
<tr>
<td>$SO(4rs)$</td>
<td>$Sp(2r) \times Sp(2s)$</td>
<td>$['D',2rs] \Rightarrow ['D',r] \times ['D',s]$</td>
</tr>
<tr>
<td>$Sp(4rs+2s)$</td>
<td>$SO(2r+1) \times Sp(2s)$</td>
<td>$['C',2rs+s] \Rightarrow ['B',r] \times ['C',s]$</td>
</tr>
<tr>
<td>$Sp(4rs)$</td>
<td>$Sp(2r) \times SO(2s)$</td>
<td>$['C',2rs] \Rightarrow ['C',r] \times ['D',s]$</td>
</tr>
</tbody>
</table>

These branching rules are obtained using rule="tensor".
Symmetric powers

The \( k \)-th symmetric and exterior power homomorphisms map \( GL(n) \to GL\left(\binom{n+k-1}{k}\right) \) and \( GL\left(\binom{n}{k}\right) \). The corresponding branching rules are not implemented but a special case is. The \( k \)-th symmetric power homomorphism \( SL(2) \to GL(k+1) \) has its image inside of \( SO(2r+1) \) if \( k = 2r \) and inside of \( Sp(2r) \) if \( k = 2r - 1 \). Hence there are branching rules:

\[
['B',r] \Rightarrow A1
['C',r] \Rightarrow A1
\]

and these may be obtained using \texttt{rule="symmetric\_power"}.

Plethysms

The above branching rules are sufficient for most cases, but a few fall between the cracks. Mostly these involve maximal subgroups of fairly small rank.

The rule \texttt{rule="plethysm"} is a powerful rule that includes any branching rule from types \( A, B, C \) or \( D \) as a special case. Thus it could be used in place of the above rules and would give the same results. However, it is most useful when branching from \( G \) to a maximal subgroup \( H \) such that \( \text{rank}(H) < \text{rank}(G) - 1 \).

We consider a homomorphism \( H \to G \) where \( G \) is one of \( SL(r+1), SO(2r+1), Sp(2r) \) or \( SO(2r) \). The function \texttt{branching\_rule\_from\_plethysm} produces the corresponding branching rule. The main ingredient is the character \( \chi \) of the representation of \( H \) that is the homomorphism to \( GL(r+1), GL(2r+1) \) or \( GL(2r) \).

Let us consider the symmetric fifth power representation of \( SL(2) \). This is implemented above by \texttt{rule="symmetric\_power"}, but suppose we want to use \texttt{rule="plethysm"}. First we construct the homomorphism by invoking its character, to be called \texttt{chi}:

\begin{verbatim}
sage: A1 = WeylCharacterRing("A1", style="coroots")
sage: chi = A1([5])
sage: chi.degree()
6
sage: chi.frobenius_schur_indicator()
-1
\end{verbatim}

This confirms that the character has degree 6 and is symplectic, so it corresponds to a homomorphism \( SL(2) \to Sp(6) \), and there is a corresponding branching rule \( C3 \to A1 \):

\begin{verbatim}
sage: A1 = WeylCharacterRing("A1", style="coroots")
sage: C3 = WeylCharacterRing("C3", style="coroots")
sage: chi = A1([5])
sage: sym5rule = branching_rule_from_plethysm(chi, "C3")
sage: [C3(hwv).branch(A1, rule=sym5rule) for hwv in C3.fundamental_weights()]
\end{verbatim}

This is identical to the results we would obtain using \texttt{rule="symmetric\_power"}:

\begin{verbatim}
sage: A1 = WeylCharacterRing("A1", style="coroots")
sage: C3 = WeylCharacterRing("C3", style="coroots")
sage: [C3(v).branch(A1, rule="symmetric\_power") for v in C3.fundamental_weights()]
\end{verbatim}

But the next example of plethysm gives a branching rule not available by other methods:
In this example, \( ad \) is the 14-dimensional adjoint representation of the exceptional group \( G_2 \). Since the Frobenius-Schur indicator is 1, the representation is orthogonal, and factors through \( SO(14) \), that is, \( D_7 \).

We do not actually have to create the character (or for that matter its ambient WeylCharacterRing) in order to create the branching rule:

```python
sage: branching_rule("D4","A2.adjoint_representation()","plethysm")
plethysm (along A2(1,1)) branching rule D4 => A2
```

The adjoint representation of any semi-simple Lie group is orthogonal, so we do not need to compute the Frobenius-Schur indicator.

### Miscellaneous other subgroups

Use \texttt{rule="miscellaneous"} for the following rules. Every maximal subgroup \( H \) of an exceptional group \( G \) are either among these, or the five \( A_1 \) subgroups described in the next section, or (if \( G \) and \( H \) have the same rank) is available using \texttt{rule="extended"}.

\[
\begin{align*}
B_3 & \to G_2, \\
E_6 & \to A_2, \\
E_6 & \to G_2, \\
F_4 & \to G_2 \times A_1, \\
E_6 & \to G_2 \times A_2, \\
E_7 & \to G_2 \times C_3, \\
E_7 & \to F_4 \times A_1, \\
E_7 & \to A_1 \times A_1, \\
E_7 & \to G_2 \times A_1, \\
E_7 & \to A_2 \\
E_8 & \to G_2 \times F_4, \\
E_8 & \to A_2 \times A_1, \\
E_8 & \to B_2
\end{align*}
\]
The first rule corresponds to the embedding of $G_2$ in $\text{SO}(7)$ in its action on the trace zero octonions. The two branching rules from $E_6$ to $G_2$ or $A_2$ are described in [Testerman1989]. We caution the reader that Theorem G.2 of that paper, proved there in positive characteristic is false over the complex numbers. On the other hand, the assumption of characteristic $p$ is not important for Theorems G.1 and A.1, which describe the torus embeddings, hence contain enough information to compute the branching rule. There are other ways of embedding $G_2$ or $A_2$ into $E_6$. These may embeddings be characterized by the condition that the two 27-dimensional representations of $E_6$ restrict irreducibly to $G_2$ or $A_2$. Their images are maximal subgroups.

The remaining rules come about as follows. Let $G$ be $F_4$, $E_6$, $E_7$, or $E_8$, and let $H$ be $G_2$, or else (if $G = E_7$) $F_4$. We embed $H$ into $G$ in the most obvious way; that is, in the chain of subgroups

$$G_2 \subset F_4 \subset E_6 \subset E_7 \subset E_8$$

Then the centralizer of $H$ is $A_1$, $A_2$, $C_3$, $F_4$ (if $H = G_2$) or $A_1$ (if $G = E_7$ and $H = F_4$). This gives us five of the cases. Regarding the branching rule $E_6 \to G_2 \times A_2$, Rubenthaler [Rubenthaler2008] describes the embedding and applies it in an interesting way.

The embedding of $A_1 \times A_1$ into $E_7$ is as follows. Deleting the 5 node of the $E_7$ Dynkin diagram gives the Dynkin diagram of $A_1 \times A_2$, so this is a Levi subgroup. We embed $\text{SL}(2)$ into this Levi subgroup via the representation $[4] \otimes [2]$. This embeds the first copy of $A_1$. The other $A_1$ is the connected centralizer. See [Seitz1991], particularly the proof of (3.12).

The embedding if $G_2 \times A_1$ into $E_7$ is as follows. Deleting the 2 node of the $E_7$ Dynkin diagram gives the $A_6$ Dynkin diagram, which is the Levi subgroup $\text{SL}(7)$. We embed $G_2$ into $\text{SL}(7)$ via the irreducible seven-dimensional representation of $G_2$. The $A_1$ is the centralizer.

The embedding if $A_2 \times A_1$ into $E_8$ is as follows. Deleting the 2 node of the $E_8$ Dynkin diagram gives the $A_7$ Dynkin diagram, which is the Levi subgroup $\text{SL}(8)$. We embed $A_2$ into $\text{SL}(8)$ via the irreducible eight-dimensional adjoint representation of $\text{SL}(2)$. The $A_1$ is the centralizer.

The embedding $A_2$ into $E_7$ is proved in [Seitz1991] (5.8). In particular, he computes the embedding of the $\text{SL}(3)$ torus in the $E_7$ torus, which is what is needed to implement the branching rule. The embedding of $B_2$ into $E_8$ is also constructed in [Seitz1991] (6.7). The embedding of the $B_2$ Cartan subalgebra, needed to implement the branching rule, is easily deduced from (10) on page 111.

### Maximal A1 subgroups of Exceptional Groups

There are seven embeddings of $\text{SL}(2)$ into an exceptional group as a maximal subgroup: one each for $G_2$ and $F_4$, two nonconjugate embeddings for $E_7$ and three for $E_8$. These are constructed in [Testerman1992]. Create the corresponding branching rules as follows. The names of the rules are roman numerals referring to the seven cases of Testerman’s Theorem 1:

```
sage: branching_rule("G2","A1","i")
i branching rule G2 => A1
sage: branching_rule("F4","A1","ii")
ii branching rule F4 => A1
sage: branching_rule("E7","A1","iii")
iii branching rule E7 => A1
sage: branching_rule("E7","A1","iv")
iv branching rule E7 => A1
sage: branching_rule("E8","A1","v")
v branching rule E8 => A1
sage: branching_rule("E8","A1","vi")
```

(continues on next page)
vi branching rule E8 => A1

```
sage: branching_rule("E8","A1","vii")
```

vii branching rule E8 => A1

The embeddings are characterized by the root restrictions in their branching rules: usually a simple root of the ambient group \(G\) restricts to the unique simple root of \(A_1\), except for root \(\alpha_4\) for rules iv, vi and vii, and the root \(\alpha_6\) for root vii; this is essentially the way Testerman characterizes the embeddings, and this information may be obtained from Sage by employing the `describe()` method of the branching rule. Thus:

```
sage: branching_rule("E8","A1","vii").describe()
```

```
0 2
|
0---0---0---0---0---0---0---0---0---0
1 3 4 5 6 7 8 0
E8-
root restrictions E8 => A1:

0
1
A1

1 => 1
2 => 1
3 => 1
4 => (zero)
5 => 1
6 => (zero)
7 => 1
8 => 1
```

For more detailed information use `verbose=True`

---

**Writing your own branching rules**

Sage has many built-in branching rules. Indeed, at least up to rank eight (including all the exceptional groups) branching rules to all maximal subgroups are implemented as built in rules, except for a few obtainable using `branching_rule_from_plethysm`. This means that all the rules in [McKayPatera1981] are available in Sage.

Still in this section we are including instructions for coding a rule by hand. As we have already explained, the branching rule is a function from the weight lattice of \(G\) to the weight lattice of \(H\), and if you supply this you can write your own branching rules.

As an example, let us consider how to implement the branching rule \(A_3 \rightarrow C_2\). Here \(H = C_2 = \text{Sp}(4)\) embedded as a subgroup in \(A_3 = \text{GL}(4)\). The Cartan subalgebra \(\text{Lie}(U)\) consists of diagonal matrices with eigenvalues \(u_1, u_2, -u_2, -u_1\). Then \(C_2.\text{space}()\) is the two dimensional vector spaces consisting of the linear functionals \(u_1\) and \(u_2\) on \(U\). On the other hand \(\text{Lie}(T) = \mathbb{R}^4\). A convenient way to see the restriction is to think of it as the adjoint of the map \([u_1,u_2] \rightarrow [u_1,u_2,-u_2,-u_1]\), that is, \([x0,x1,x2,x3] \rightarrow [x0-x3,x1-x2]\). Hence we may encode the rule:

```python
def brule(x):
    return [x[0]-x[3], x[1]-x[2]]
```
or simply:

```
brule = lambda x: [x[0]-x[3], x[1]-x[2]]
```

Let us check that this agrees with the built-in rule:

```
sage: A3 = WeylCharacterRing(['A', 3])
sage: C2 = WeylCharacterRing(['C', 2])
sage: brule = lambda x: [x[0]-x[3], x[1]-x[2]]
sage: A3(1,1,0,0).branch(C2, rule=brule)
C2(0,0) + C2(1,1)
sage: A3(1,1,0,0).branch(C2, rule="symmetric")
C2(0,0) + C2(1,1)
```

Although this works, it is better to make the rule into an element of the `BranchingRule` class, as follows.

```
sage: brule = BranchingRule("A3","C2",lambda x : [x[0]-x[3], x[1]-x[2]],"custom")
sage: A3(1,1,0,0).branch(C2, rule=brule)
C2(0,0) + C2(1,1)
```

**Automorphisms and triality**

The case where $G = H$ can be treated as a special case of a branching rule. In most cases if $G$ has a nontrivial outer automorphism, it has order two, corresponding to the symmetry of the Dynkin diagram. Such an involution exists in the cases $A_r, D_r, E_6$.

So the automorphism acts on the representations of $G$, and its effect may be computed using the branching rule code:

```
sage: A4 = WeylCharacterRing("A4",style="coroots")
sage: A4(1,0,1,0).degree()
45
sage: A4(0,1,0,1).degree()
45
sage: A4(1,0,1,0).branch(A4,rule="automorphic")
A4(0,1,0,1)
```

In the special case where $G=D_4$, the Dynkin diagram has extra symmetries, and these correspond to outer automorphisms of the group. These are implemented as the "triality" branching rule:

```
sage: branching_rule("D4","D4","triality").describe()
```

(continues on next page)
| 0---0---0 |
| 1 2 3 |
D4
1 => 3
2 => 2
3 => 4
4 => 1

For more detailed information use verbose=True

Triality is not an automorphism of $SO(8)$, but of its double cover $spin(8)$. Note that $spin(8)$ has three representations of degree 8, namely the standard representation of $SO(8)$ and the two eight-dimensional spin representations. These are permuted by triality:

```
sage: D4 = WeylCharacterRing("D4",style="coroots")
sage: D4(0,0,0,1).branch(D4,rule="triality")
D4(1,0,0,0)
sage: D4(0,0,0,1).branch(D4,rule="triality").branch(D4,rule="triality")
D4(0,0,1,0)
sage: D4(0,0,0,1).branch(D4,rule="triality").branch(D4,rule="triality").branch(D4,rule="triality")
D4(0,0,0,1)
```

By contrast, rule="automorphic" simply interchanges the two spin representations, as it always does in type $D$:

```
sage: D4(0,0,0,1).branch(D4,rule="automorphic")
D4(0,0,1,0)
sage: D4(0,0,1,0).branch(D4,rule="automorphic")
D4(0,0,0,1)
```

Weyl Groups, Coxeter Groups and the Bruhat Order

Classical and affine Weyl groups

You can create Weyl groups and affine Weyl groups for any root system. A variety of methods are available for these. Some of these are methods are available for general Coxeter groups.

By default, elements of the Weyl group are represented as matrices:

```
sage: W = WeylGroup("A3",prefix="s")
sage: [s1,s2,s3] = W.simple_reflections()
```

You may prefer a notation in which elements are written out as products of simple reflections. In order to implement this you need to specify a prefix, typically "s":

```
sage: W = WeylGroup("A3",prefix="s")
sage: [s1,s2,s3] = W.simple_reflections()
```
sage: (s1*s2*s1).length()  
3

sage: W.long_element()

s1*s2*s3*s1*s2*s1

sage: s1*s2*s3*s1*s2*s1 == s3*s2*s1*s3*s2*s3

True

The Weyl group acts on the ambient space of the root lattice, which is accessed by the method domain. To illustrate this, recall that if $w_0$ is the long element then $\alpha \mapsto -w_0(\alpha)$ is a permutation of the simple roots. We may compute this as follows:

sage: W = WeylGroup("E6", prefix="s")
sage: w0 = W.long_element(); w0

s1*s3*s4*s5*s6*s2*s4*s5*s3*s4*s1*s3*s2*s4*s5*s3*s4*s1*s3*s2*s4*s5*s3*s4*s1*s3*s2*s4*s1*s3*s2*s1

sage: sr = W.domain().simple_roots().list(); sr

[(1/2, -1/2, -1/2, -1/2, -1/2, -1/2), (1, 1, 0, 0, 0, 0, 0, 0), (0, -1, 1, 0, 0, 0, 0, 0), (0, 0, -1, 1, 0, 0, 0, 0), (0, 0, 0, -1, 1, 0, 0, 0)]

sage: [-w0.action(a) for a in sr]

[(0, 0, 0, -1, 1, 0, 0, 0), (1, 1, 0, 0, 0, 0, 0, 0), (0, -1, 1, 0, 0, 0, 0, 0), (0, 0, -1, 1, 0, 0, 0, 0), (0, 0, 0, -1, 1, 0, 0, 0), (1/2, -1/2, -1/2, -1/2, -1/2, -1/2)]

We may ask when this permutation is trivial. If it is nontrivial it induces an automorphism of the Dynkin diagram, so it must be nontrivial when the Dynkin diagram has no automorphism. But if there is a nontrivial automorphism, the permutation might or might not be trivial:

sage: def roots_not_permuted(ct):
    ....:     W = WeylGroup(ct)
    ....:     w0 = W.long_element()
    ....:     sr = W.domain().simple_roots()
    ....:     return all(a == -w0.action(a) for a in sr)

sage: for ct in [CartanType(['D', r]) for r in [2..8]]:
    ....:     print("{} {}\n".format(ct, roots_not_permuted(ct)))

['D', 2] True
['D', 3] False
['D', 4] True
['D', 5] False
['D', 6] True
['D', 7] False
['D', 8] True

If $\alpha$ is a root let $r_\alpha$ denote the reflection in the hyperplane that is orthogonal to $\alpha$. We reserve the notation $s_\alpha$ for the simple reflections, that is, the case where $\alpha$ is a simple root. The reflections are just the conjugates of the simple reflections.

The reflections are the values in a finite family, which is a wrapper around a python dictionary. The keys are the positive roots, so given a positive root, you can look up the corresponding reflection. If you want a list of all reflections, you can use the usual methods to construct a list (e.g., using the list function) or use the method values for the family of reflections:

sage: W = WeylGroup("B3", prefix="s")
sage: ref = W.reflections(); ref

(continues on next page)
Finite family \{(1, -1, 0): s1, (0, 1, -1): s2, ...\}

```
sage: \[a1,a2,a3\] = W.domain().simple_roots()
sage: a1+a2+a3
(1, 0, 0)
sage: ref[a1+a2+a3]
s1*s2*s3*s2*s1

sage: sorted(ref)
[s1*s2*s3*s2*s1,
s2*s3*s1*s2*s3*s1*s2,
s3*s1*s2*s3*s1,
s1*s2*s1,
s1,
s2*s3*s2,
s3*s2*s3,
s2,
s3]
```

If instead you want a family whose keys are the reflections and whose values are the roots, you may use the inverse family:

```
sage: from pprint import pprint
sage: W = WeylGroup("B3",prefix="s")
sage: [s1,s2,s3] = W.simple_reflections()
sage: altref = W.reflections().inverse_family()
sage: altref
Finite family \{s1*s2*s3*s2*s1: (1, 0, 0),
s2*s3*s1*s2*s3*s1*s2: (1, 1, 0),
s3*s1*s2*s3*s1: (1, 0, 1),
s1*s2*s1: (1, 0, -1),
s1: (1, -1, 0),
s2*s3*s2: (0, 1, 0),
s3*s2*s3: (0, 1, 1),
s2: (0, 1, -1),
s3: (0, 0, 1)\}
sage: altref[s3*s2*s3]
(0, 1, 1)
```

**Note:** The behaviour of this function was changed in trac ticket #20027.

The Weyl group is implemented as a GAP matrix group. You therefore can display its character table. The character table is returned as a string, which you can print:

```
sage: print(WeylGroup("D4").character_table())
CT1
   2   6   4   5   1   3   5   5   4   3   3   1   4   6
   3   1   .   .   1   .   .   .   .   .   1   .   1

   1a  2a  2b  6a  4a  2c  2d  2e  4b  4c  3a  4d  2f

X.1  1   1   1   1   1   1   1   1   1   1   1   1   1
```

(continues on next page)
### Affine Weyl groups

Affine Weyl groups may be created the same way. You simply begin with an affine Cartan type:

```python
sage: W = WeylGroup(['A',2,1],prefix="s")
sage: W.cardinality()
+Infinity
sage: [s0,s1,s2] = W.simple_reflections()
```

The affine Weyl group differs from a classical Weyl group since it is infinite. The associated classical Weyl group is a subgroup that may be extracted as follows:

```python
sage: W = WeylGroup(['A',2,1],prefix="s")
sage: W1 = W.classical(); W1
Parabolic Subgroup of the Weyl Group of type ['A', 2, 1] (as a matrix group acting on the root space)
sage: W1.simple_reflections()
Finite family {1: s1, 2: s2}
```

Although $W_1$ in this example is isomorphic to $\text{WeylGroup}('A2')$ it has a different matrix realization:

```python
sage: for s in WeylGroup(['A',2,1]).classical().simple_reflections():
    ....:     print(s)
    ....:     print('')
    [ 1 0 0]
    [ 1 -1 1]
    [ 0 0 1]

sage: for s in WeylGroup(['A',2]).simple_reflections():
    ....:     print(s)
    ....:     print('')
    [0 1 0]
```

(continues on next page)
Bruhat order

The Bruhat partial order on the Weyl group may be defined as follows.

If \( u, v \in W \), find a reduced expression of \( v \) into a product of simple reflections: \( v = s_1 \cdots s_n \). (It is not assumed that the \( s_i \) are distinct.) If omitting some of the \( s_i \) gives a product that represents \( u \), then \( u \leq v \).

The Bruhat order is implemented in Sage as a method of Coxeter groups, and so it is available for Weyl groups, classical or affine.

If \( u, v \in W \) then \( u.\text{bruhat\_le}(v) \) returns True if \( u \leq v \) in the Bruhat order.

If \( u \leq v \) then the Bruhat interval \([u, v]\) is defined to be the set of all \( t \) such that \( u \leq t \leq v \). One might try to implement this as follows:

```python
sage: W = WeylGroup("A2",prefix="s")
sage: [s1,s2] = W.simple_reflections()
sage: def bi(u,v) : return [t for t in W if u.\text{bruhat\_le}(t) and t.\text{bruhat\_le}(v)]
...  
sage: bi(s1,s1*s2*s1)
[s1*s2*s1, s2*s1, s1, s1*s2*s1]
```

This would not be a good definition since it would fail if \( W \) is affine and be inefficient of \( W \) is large. Sage has a Bruhat interval method:

```python
sage: W = WeylGroup("A2",prefix="s")
sage: [s1,s2] = W.simple_reflections()
sage: W.\text{bruhat\_interval}(s1,s1*s2*s1)
[s1*s2*s1, s2*s1, s1*s2*s1, s1]
```

This works even for affine Weyl groups.

The Bruhat graph

The Bruhat graph is a structure on the Bruhat interval. Suppose that \( u \leq v \). The vertices of the graph are \( x \) with \( u \leq x \leq v \). There is a vertex connecting \( x, y \in [x, y] \) if \( x = y \cdot r \) where \( r \) is a reflection. If this is true then either \( x < y \) or \( y < x \).

If \( W \) is a classical Weyl group the Bruhat graph is implemented in Sage:
sage: W = WeylGroup("A3",prefix="s")
sage: [s1,s2,s3] = W.simple_reflections()

sage: bg = W.bruhat_graph(s2,s2*s1*s3*s2); bg
Digraph on 10 vertices

sage: bg.show3d()

The Bruhat graph has interesting regularity properties that were investigated by Carrell and Peterson. It is a regular
graph if both the Kazhdan Lusztig polynomials $P_{u,v}$ and $P_{w_0,v,w_0u}$ are 1, where $w_0$ is the long Weyl group element. It is
closely related to the Deodhar conjecture, which was proved by Deodhar, Carrell and Peterson, Dyer and Polo.

Deodhar proved that if $u < v$ then the Bruhat interval $[u,v]$ contains as many elements of odd length as it does of even
length. We observe that often this can be strengthened: If there exists a reflection $r$ such that left (or right) multiplication
by $r$ takes the Bruhat interval $[u,v]$ to itself, then this gives an explicit bijection between the elements of odd and even
length in $[u,v]$.

Let us search for such reflections. Put the following commands in a file and load the file:

```python
W = WeylGroup("A3",prefix="s")
[s1,s2,s3] = W.simple_reflections()
ref = W.reflections().keys()
def find_reflection(u,v):
    bi = W.bruhat_interval(u,v)
    ret = []
    for r in ref:
        if all( r*x in bi for x in bi):
            ret.append(r)
    return ret

for v in W:
    for u in W.bruhat_interval(1,v):
        if u != v:
            print((u,v,find_reflection(u,v)))
```

This shows that the Bruhat interval is stabilized by a reflection for all pairs $(u,v)$ with $u < v$ except the following two:
$s_3s_1, s_1s_2s_3s_2s_1$ and $s_2, s_2s_3s_1s_2$. Now these are precisely the pairs such that $u \prec v$ in the notation of Kazhdan and
Lusztig, and $l(v) - l(u) > 1$. One should not rashly suppose that this is a general characterization of the pairs $(u,v)$
such that no reflection stabilizes the Bruhat interval, for this is not true. However it does suggest that the question is
worthy of further investigation.
Classical Crystals

A classical crystal is one coming from the finite (classical) types $A_r, B_r, C_r, D_r, E_{6,7,8}, F_4$, and $G_2$. Here we describe some background before going into the general theory of crystals and the type dependent combinatorics.

Tableaux and representations of $GL(n)$

Let $\lambda$ be a partition. The Young diagram of $\lambda$ is the array of boxes having $\lambda_i$ boxes in the $i$-th row, left adjusted. Thus if $\lambda = (3, 2)$ the diagram is:

```

```

A semi-standard Young tableau of shape $\lambda$ is a filling of the box by integers in which the rows are weakly increasing and the columns are strictly increasing. Thus

```
1 2 2
2 3
```

is a semistandard Young tableau. Sage has a Tableau class, and you may create this tableau as follows:

```
sage: T = Tableau([[1, 2, 2], [2, 3]]); T

```

A partition of length $\leq r+1$ is a dominant weight for $GL(r+1, \mathbb{C})$ according to the description of the ambient space in Standard realizations of the ambient spaces. Therefore it corresponds to an irreducible representation $\pi_\lambda = \pi_{GL(r+1)}^{\lambda}$ of $GL(r+1, \mathbb{C})$.

It is true that not every dominant weight $\lambda$ is a partition, since a dominant weight might have some values $\lambda_i$ negative. The dominant weight $\lambda$ is a partition if and only if the character of $\lambda$ is a polynomial as a function on the space $Mat_n(\mathbb{C})$. Thus for example $\det^{-1} = \pi_\lambda$ with $\lambda = (-1, \ldots, -1)$, which is a dominant weight but not a partition, and the character is not a polynomial function on $Mat_n(\mathbb{C})$.

**Theorem** [Littlewood] If $\lambda$ is a partition, then the number of semi-standard Young tableaux with shape $\lambda$ and entries in $\{1, 2, \ldots, r+1\}$ is the dimension of $\pi_\lambda$.

For example, if $\lambda = (3, 2)$ and $r = 2$, then we find 15 tableaux with shape $\lambda$ and entries in $\{1, 2, 3\}$:

```
1 1 1
2 2

1 1 2
2 2

1 1 3
2 3

1 2 2
2 3

1 1 2
2 3

1 2 2
2 3

1 1 1
2 3

1 2 2
2 3

1 2 2
2 3

This is consistent with the theorem since the dimension of the irreducible representation of $GL(3)$ with highest weight $(3, 2, 0)$ has dimension 15:
In fact we may obtain the character of the representation from the set of tableaux. Indeed, one of the definitions of the Schur polynomial (due to Littlewood) is the following combinatorial one. If $T$ is a tableau, define the weight of $T$ to be $\text{wt}(T) = (k_1, \ldots, k_n)$ where $k_i$ is the number of $i$’s in the tableaux. Then the multiplicity of $\mu$ in the character $\chi_\lambda$ is the number of tableaux of weight $\lambda$. Thus if $z = (z_1, \ldots, z_n)$, we have

$$\chi_\lambda(z) = \sum_T z^{\text{wt}(T)}$$

where the sum is over all semi-standard Young tableaux of shape $\lambda$ that have entries in $\{1, 2, \ldots, r + 1\}$.

**Frobenius-Schur Duality**

Frobenius-Schur duality is a relationship between the representation theories of the symmetric group and general linear group. We will relate this to tableaux in the next section.

Representations of the symmetric group $S_k$ are parametrized by partitions $\lambda$ of $k$. The parametrization may be characterized as follows. Let $n$ be any integer $\geq k$. Then both $GL(n, \mathbb{C})$ and $S_k$ act on $\otimes^k V$ where $V = \mathbb{C}^n$. Indeed, $GL(n)$ acts on each $V$ and $S_k$ permutes them. Then if $\pi^{GL(n)}_\lambda$ is the representation of $GL(n, \mathbb{C})$ with highest weight vector $\lambda$ and $\pi^{S_k}_\lambda$ is the irreducible representation of $S_k$ parametrized by $\lambda$ then

$$\otimes^k V \cong \bigoplus_{\lambda \vdash k} \pi^{GL(n)}_\lambda \otimes \pi^{S_k}_\lambda$$

as bimodules for the two groups. This is Frobenius-Schur duality and it serves to characterize the parametrization of the irreducible representations of $S_k$ by partitions of $k$.

**Counting pairs of tableaux**

In both the representation theory of $GL(n)$ and the representation theory of $S_k$, the degrees of irreducible representations can be expressed in terms of the number of tableaux of the appropriate type. We have already stated the theorem for $GL(n)$. For $S_k$, it goes as follows.

Let us say that a semistandard Young tableau $T$ of shape $\lambda \vdash k$ is standard if $T$ contains each entry $1, 2, \ldots, k$ exactly once. Thus both rows and columns are strictly increasing.

**Theorem** [Young, 1927] The degree of $\pi_\lambda$ is the number of standard tableaux of shape $\lambda$.

Now let us consider the implications of Frobenius-Schur duality. The dimension of $\otimes^k V$ is $n^k$. Therefore $n^k$ is equal to the number of pairs $(T_1, T_2)$ of tableaux of the same shape $\lambda \vdash k$, where the first tableau is standard (in the alphabet $1, 2, \ldots, k$), and the second the second semistandard (in the alphabet $1, 2, \ldots, n$).
The Robinson-Schensted-Knuth correspondence

The last purely combinatorial statement has a combinatorial proof, based on the Robinson-Schensted-Knuth (RSK) correspondence.

References:

- [Knuth1998], section “Tableaux and Involutions”.
- [Knuth1970]
- [Fulton1997]
- [Stanley1999]

The RSK correspondence gives bijections between pairs of tableaux of various types and combinatorial objects of different types. We will not review the correspondence in detail here, but see the references. We note that Schensted insertion is implemented as the method `schensted_insertion` of Tableau class in Sage.

Thus we have the following bijections:

- Pairs of tableaux $T_1$ and $T_2$ of shape $\lambda$ where $\lambda$ runs through the partitions of $k$ such that $T_1$ is a standard tableau and $T_2$ is a semistandard tableau in $1, 2, \ldots, n$ are in bijection with the $n^k$ words of length $k$ in $1, 2, \ldots, n$.
- Pairs of standard tableaux of the same shape $\lambda$ as $\lambda$ runs through the partitions of $k$ are in bijection with the $k!$ elements of $S_k$.
- Pairs of tableaux $T_1$ and $T_2$ of the same shape $\lambda$ but arbitrary size in $1, 2, 3, \ldots, n$ are in bijection with $n \times n$ positive integer matrices.
- Pairs of tableaux $T_1$ and $T_2$ of conjugate shapes $\lambda$ and $\lambda'$ are in bijection with $n \times n$ matrices with entries 0 or 1.

The second of these four bijection gives a combinatorial proof of the fact explained above, that the number of pairs $(T_1, T_2)$ of tableaux of the same shape $\lambda \vdash k$, where the first tableau is standard (in the alphabet $1, 2, \ldots, k$), and the second the second semistandard (in the alphabet $1, 2, \ldots, n$). So this second bijection is a combinatorial analog of Frobenius-Schur duality.

Analogies between representation theory and combinatorics

The four combinatorial bijections (variants of RSK) cited above have the following analogs in representation theory.

- The first combinatorial fact corresponds to Frobenius-Schur duality, as we have already explained.
- The second combinatorial fact also has an analog in representation theory. The group algebra $\mathbb{C}[S_k]$ is an $S_k \times S_k$ bimodule with of dimension $k!$. It decomposes as a direct sum of $\pi_{\lambda}^{S_k} \otimes \pi_{\lambda}^{S_k}$. Both the combinatorial fact and the decomposition of $\mathbb{C}[S_k]$ show that the number of pairs of standard tableaux of size $k$ and the same shape equals $k!$.
- The third combinatorial fact is analogous to the decomposition of the ring of polynomial functions on $\text{Mat}(n, \mathbb{C})$ on which $GL(n, \mathbb{C}) \times GL(n, \mathbb{C})$ acts by $(g_1, g_2)f(X) = f(g_1Xg_2)$. The polynomial ring decomposes into the direct sum of $\pi_{\lambda}^{GL(n)} \otimes \pi_{\lambda}^{GL(n)}$.
- The last combinatorial fact is analogous to the decomposition of the exterior algebra over $\text{Mat}(n, \mathbb{C})$.

Taking traces gives the Cauchy identity:

$$\sum_{\lambda} s_\lambda(x_1, \ldots, x_n)s_\lambda(y_1, \ldots, y_n) = \prod_{i,j}(1 - x_i y_j)^{-1}$$

where $x_i$ are the eigenvalues of $g_1$ and $y_j$ are the eigenvalues of $g_2$. The sum is over all partitions $\lambda$. 

- The last combinatorial fact is analogous to the decomposition of the exterior algebra over $\text{Mat}(n, \mathbb{C})$.
Taking traces gives the dual Cauchy identity:

$$\sum_{\lambda} s_{\lambda}(x_1, \cdots, x_n)s_{\lambda'}(y_1, \cdots, y_n) = \prod_{i,j}(1 + x_iy_j).$$

Again the sum is over partitions \(\lambda\) and here \(\lambda'\) is the conjugate partition.

**Interpolating between representation theory and combinatorics**

The theory of quantum groups interpolates between the representation theoretic picture and the combinatorial picture, and thereby explains these analogies. The representation \(\pi_{GL(n)}^\lambda\) is reinterpreted as a module for the quantized enveloping algebra \(U_q(g)\), and the representation \(\pi_{\lambda}^\lambda\) is reinterpreted as a module for the Iwahori Hecke algebra. Then Frobenius-Schur duality persists. See [Jimbo1986]. When \(q \to 1\), we recover the representation story. When \(q \to 0\), we recover the combinatorial story.

**Kashiwara crystals**

References:
- [Kashiwara1995]
- [KashiwaraNakashima1994]
- [HongKang2002]

Kashiwara considered the highest weight modules of quantized enveloping algebras \(U_q(G)\) in the limit when \(q \to 0\). The enveloping algebra cannot be defined when \(q = 0\), but a limiting structure can still be detected. This is the crystal basis of the module.

Kashiwara’s crystal bases have a combinatorial structure that sheds light even on purely combinatorial constructions on tableaux that predated quantum groups. It gives a good generalization to other Cartan types (or more generally to Kac-Moody algebras).

Let \(\Lambda\) be the weight lattice of a Cartan type with root system \(\Phi\). We now define a crystal of type \(\Phi\). Let \(B\) be a set, and let \(0 \notin B\) be an auxiliary element. For each index \(1 \leq i \leq r\) we assume there given maps \(e_i, f_i : B \to B \cup \{0\}\), maps \(\varepsilon_i, \varphi_i : B \to \mathbb{Z}\) and a map \(\text{wt} : B \to \Lambda\) satisfying certain assumptions, which we now describe. It is assumed that if \(x, y \in B\) then \(e_i(x) = y\) if and only if \(f_i(y) = x\). In this case, it is assumed that

$$\text{wt}(y) = \text{wt}(x) + \alpha_i, \quad \varepsilon_i(x) = \varepsilon_i(y) + 1, \quad \varphi_i(x) = \varphi_i(y) - 1.$$  

Moreover, we assume that

$$\varphi_i(x) - \varepsilon_i(x) = \langle \text{wt}(x), \alpha_i^\vee \rangle$$

for all \(x \in B\).

We call a crystal regular if it satisfies the additional assumption that \(\varepsilon_i(v)\) is the number of times that \(e_i\) may be applied to \(v\), and that \(\varphi_i(v)\) is the number of times that \(f_i\) may be applied. That is, \(\varphi_i(x) = \max\{k | f_i^k(x) \neq 0\}\) and \(\varepsilon_i(x) = \max\{k | e_i^k(x) \neq 0\}\). Kashiwara also allows \(\varepsilon_i\) and \(\varphi_i\) to take the value \(-\infty\).

**Note:** Most of the crystals that we are concerned with here are regular.

Given the crystal \(B\), the character \(\chi_B\) is:

$$\sum_{v \in B} z^{\text{wt}(v)}.$$

12.1. Thematic tutorial document tree 175
Given any highest weight $\lambda$, constructions of Kashiwara and Nakashima, Littelmann and others produce a crystal $\chi_{B_\lambda}$ such that $\chi_{B_\lambda} = \chi_\lambda$, where $\chi_\lambda$ is the irreducible character with highest weight $\lambda$, as in *Representations and characters*. The crystal $B_\lambda$ is not uniquely characterized by the properties that we have stated so far. For Cartan types $A, D, E$ (more generally, any simply-laced type) it may be characterized by these properties together with certain other Stembridge axioms. We will take it for granted that there is a unique “correct” crystal $B_\lambda$ and discuss how these are constructed in Sage.

### Installing dot2tex

Before giving examples of crystals, we digress to help you install dot2tex, which you will need in order to make latex images of crystals.

*dot2tex* is an optional package of sage and the latest version can be installed via:

```
$ sage -i dot2tex
```

This also requires various LaTeX packages. The following command checks whether those are available:

```
sage: from sage.graphs.graph_latex import check_tkz_graph
sage: check_tkz_graph()  # random
```

### Crystals of tableaux in Sage

All crystals that are currently in Sage can be accessed by `crystals.<tab>`.

For type $A_r$, Kashiwara and Nakashima put a crystal structure on the set of tableaux with shape $\lambda$ in $1, 2, \ldots, r + 1$, and this is a realization of $B_\lambda$. Moreover, this construction extends to other Cartan types, as we will explain. At the moment, we will consider how to draw pictures of these crystals.

Once you have dot2tex installed, you may make images pictures of crystals with a command such as this:

```
sage: fn = tmp_filename(ext=".tex")
sage: crystals.Tableaux("A2", shape=[2,1]).latex_file(fn)
```

Here $\lambda = (2, 1) = (2, 1, 0)$. The crystal $C$ is $B_\lambda$. The character $\chi_\lambda$ will therefore be the eight-dimensional irreducible character with this highest weight. Then you may run pdflatex on the file `a2rho.tex`. This can also be achieved without the detour of saving the latex file via:

```
sage: B = crystals.Tableaux(["A",2], shape=[2,1])
sage: view(B, tightpage=True)  # optional - dot2tex graphviz, not tested (opens external window)
```

This produces the crystal graph:
You may also wish to color the edges in different colors by specifying further latex options:

```python
sage: B = crystals.Tableaux(['A',2], shape=[2,1])
sage: G = B.digraph()
sage: G.set_latex_options(color_by_label = {1:"red", 2:"yellow"})
sage: view(G, tightpage=True) # optional - dot2tex graphviz, not tested (opens external window)
```

As you can see, the elements of this crystal are exactly the eight tableaux of shape \( \lambda \) with entries in \{1,2,3\}. The convention is that if \( x, y \in B \) and \( f_i(x) = y \), or equivalently \( e_i(y) = x \), then we draw an arrow from \( x \rightarrow y \). Thus the highest weight tableau is the one with no incoming arrows. Indeed, this is:

```
1 1
2
```

We recall that the weight of the tableau is \((k_1, k_2, k_3)\) where \( k_i \) is the number of \( i \)'s in the tableau, so this tableau has weight \((2, 1, 0)\), which indeed equals \( \lambda \).

Once the crystal is created, you have access to the ambient space and its methods through the method `weight_lattice_realization()`:

```python
sage: C = crystals.Tableaux("A2", shape=[2,1])
sage: L = C.weight_lattice_realization(); L
Ambient space of the Root system of type ['A', 2]
sage: L.fundamental_weights()
Finite family {1: (1, 0, 0), 2: (1, 1, 0)}
```

The highest weight vector is available as follows:

```python
sage: C = crystals.Tableaux("A2", shape=[2,1])
sage: v = C.highest_weight_vector(); v
[[1, 1], [2]]
```

or more simply:

```
1 1
2
```
sage: C = crystals.Tableaux("A2", shape=[2,1])
sage: C[0]
[[1, 1], [2]]

Now we may apply the operators $e_i$ and $f_i$ to move around in the crystal:

sage: C = crystals.Tableaux("A2", shape=[2,1])
sage: v = C.highest_weight_vector()
sage: v.f(1)
[[1, 2], [2]]
sage: v.f(1).f(1) is None
True
sage: v.f(1).f(2)
[[1, 3], [2]]
sage: v.f(1).f(2).f(2)
[[1, 3], [3]]
sage: v.f(1).f(2).f(2).f(1)
[[2, 3], [3]]
sage: v.f(1).f(2).f(2).f(1) == v.f(2).f(1).f(1).f(2)
True

You can construct the character if you first make a Weyl character ring:

sage: A2 = WeylCharacterRing("A2")
sage: C = crystals.Tableaux("A2", shape=[2,1])
sage: C.character(A2)
A2(2,1,0)

**Crystals of letters**

For each of the classical Cartan types there is a standard crystal $B_{\text{standard}}$ from which other crystals can be built up by taking tensor products and extracting constituent irreducible crystals. This procedure is sufficient for Cartan types $A_r$ and $C_r$. For types $B_r$ and $D_r$ the standard crystal must be supplemented with spin crystals. See [KashiwaraNakashima1994] or [HongKang2002] for further details.

Here is the standard crystal of type $A_r$.

You may create the crystal and work with it as follows:

sage: C = crystals.Letters("A6")
sage: v0 = C.highest_weight_vector(); v0
1
sage: v0.f(1)
2
sage: v0.f(1).f(2)
3
sage: [v0.f(1).f(2).f(x) for x in [1..6]]
[None, None, 4, None, None, None]
sage: [v0.f(1).f(2).e(x) for x in [1..6]]
[None, 2, None, None, None, None]
Here is the standard crystal of type $B_r$.

\[
\begin{array}{cccccccc}
1 & 1 & 2 & 2 & \ldots & r & r & 0 & \vdots
\end{array}
\]

There is, additionally, a spin crystal for $B_r$, corresponding to the $2^r$-dimensional spin representation. We will not draw it, but we will describe it. Its elements are vectors $\epsilon_1 \otimes \cdots \otimes \epsilon_r$, where each spin $\epsilon_i = \pm$.

If $i < r$, then the effect of the operator $f_i$ is to annihilate $v = \epsilon_1 \otimes \cdots \otimes \epsilon_r$ unless $\epsilon_i \otimes \epsilon_{i+1} = + \otimes -$. If this is so, then $f_i(v)$ is obtained from $v$ by replacing $\epsilon_i \otimes \epsilon_{i+1}$ by $- \otimes +$. If $i = r$, then $f_r$ annihilates $v$ unless $\epsilon_r = +$, in which case it replaces $\epsilon_r$ by $-$. 

Create the spin crystal as follows. The crystal elements are represented in the signature representation listing the $\epsilon_i$:

```
sage: C = crystals.Spins("B3")
sage: C.list()
[+++; ++-, ++-, -++, ++-, --+, --+, ---]
```

Here is the standard crystal of type $C_r$.

\[
\begin{array}{cccccccc}
1 & 1 & 2 & 2 & \ldots & r & r & 0 & \vdots
\end{array}
\]

Here is the standard crystal of type $D_r$.

\[
\begin{array}{cccccccc}
1 & 1 & 2 & 2 & \ldots & r & r & 0 & \vdots
\end{array}
\]

There are two spin crystals for type $D_r$. Each consists of $\epsilon_1 \otimes \cdots \otimes \epsilon_r$ with $\epsilon_i = \pm$, and the number of spins either always even or always odd. We will not describe the effect of the root operators $f_i$, but you are invited to create them and play around with them to guess the rule:

```
sage: Cplus = crystals.SpinsPlus("D4")
sage: Cminus = crystals.SpinsMinus("D4")
```

It is also possible to construct the standard crystal for type $G_2$, $E_6$, and $E_7$. Here is the one for type $G_2$ (corresponding to the representation of degree 7):

\[
\begin{array}{cccccccc}
1 & 1 & 2 & 2 & \ldots & r & r & 0 & \vdots
\end{array}
\]

The crystal of letters is a special case of the crystal of tableaux in the sense that $B_{\text{standard}}$ is isomorphic to the crystal of tableaux whose highest weight $\lambda$ is the highest weight vector of the standard representation. Thus compare:

```
sage: crystals.Letters("A3")
The crystal of letters for type ['A', 3]
sage: crystals.Tableaux("A3", shape=[1])
The crystal of tableaux of type ['A', 3] and shape(s) [[1]]
```

These two crystals are different in implementation, but they are isomorphic. In fact the second crystal is constructed from the first. We can test isomorphisms between crystals as follows:

```
sage: Cletter = crystals.Letters(['A', 3])
sage: Ctableaux = crystals.Tableaux(['A', 3], shape=[1])
sage: Cletter.is_isomorphic(Ctableaux)
```

(continues on next page)
True
sage: Cletter.digraph().is_isomorphic(Tableaux.digraph(), certificate = True)
(True, {1: [[1]], 2: [[2]], 3: [[3]], 4: [[4]]})

where in the last step the explicit map between the vertices of the crystals is given.

Crystals of letters have a special role in the theory since they are particularly simple, yet as Kashiwara and Nakashima showed, the crystals of tableaux can be created from them. We will review how this works.

Tensor products of crystals

Kashiwara defined the tensor product of crystals in a purely combinatorial way. The beauty of this construction is that it exactly parallels the tensor product of crystals of representations. That is, if $\lambda$ and $\mu$ are dominant weights, then $B_\lambda \otimes B_\mu$ is a (usually disconnected) crystal, which may contain multiple copies of $B_\nu$ (for another dominant weight $\nu$), and the number of copies of $B_\nu$ is exactly the multiplicity of $\chi_\nu$ in $\chi_\lambda \chi_\mu$.

We will describe two conventions for the tensor product of crystals.

Kashiwara’s definition

As a set, the tensor product $B \otimes C$ of crystals $B$ and $C$ is the Cartesian product, but we denote the ordered pair $(x, y)$ with $x \in B$ and $y \in C$ by $x \otimes y$. We define $\text{wt}(x \otimes y) = \text{wt}(x) + \text{wt}(y)$. We define

$$f_i(x \otimes y) = \begin{cases} f_i(x) \otimes y & \text{if } \varphi_i(x) > \varepsilon_i(y), \\ x \otimes f_i(y) & \text{if } \varphi_i(x) \leq \varepsilon_i(y), \end{cases}$$

and

$$e_i(x \otimes y) = \begin{cases} e_i(x) \otimes y & \text{if } \varphi_i(x) \geq \varepsilon_i(y), \\ x \otimes e_i(y) & \text{if } \varphi_i(x) < \varepsilon_i(y). \end{cases}$$

It is understood that $x \otimes 0 = 0 \otimes x = 0$. We also define:

$$\varphi_i(x \otimes y) = \max(\varphi_i(y), \varphi_i(x) + \varphi_i(y) - \varepsilon_i(y)),$$

$$\varepsilon_i(x \otimes y) = \max(\varepsilon_i(y), \varepsilon_i(x) + \varepsilon_i(y) - \varphi_i(x)).$$

Alternative definition

As a set, the tensor product $B \otimes C$ of crystals $B$ and $C$ is the Cartesian product, but we denote the ordered pair $(y, x)$ with $y \in B$ and $x \in C$ by $y \otimes x$. We define $\text{wt}(x \otimes y) = \text{wt}(y) + \text{wt}(x)$. We define

$$f_i(x \otimes y) = \begin{cases} f_i(x) \otimes y & \text{if } \varphi_i(y) \leq \varepsilon_i(x), \\ x \otimes f_i(y) & \text{if } \varphi_i(y) > \varepsilon_i(x), \end{cases}$$

and

$$e_i(x \otimes y) = \begin{cases} e_i(x) \otimes y & \text{if } \varphi_i(y) < \varepsilon_i(x), \\ x \otimes e_i(y) & \text{if } \varphi_i(y) \geq \varepsilon_i(x). \end{cases}$$
It is understood that $y \otimes 0 = 0 \otimes y = 0$. We also define
\[
\varphi_i(x \otimes y) = \max(\varphi_i(x), \varphi_i(y) + \varepsilon_i(x)),
\]
\[
\varepsilon_i(x \otimes y) = \max(\varepsilon_i(y), \varepsilon_i(y) + \varepsilon_i(x) - \varphi_i(y)).
\]
The tensor product is associative: $(x \otimes y) \otimes z \mapsto x \otimes (y \otimes z)$ is an isomorphism $(B \otimes C) \otimes D \rightarrow B \otimes (C \otimes D)$, and so we may consider tensor products of arbitrary numbers of crystals.

### The relationship between the two definitions

The relationship between the two definitions is simply that the Kashiwara tensor product $B \otimes C$ is the alternate tensor product $C \otimes B$ in reverse order. Sage uses the alternative tensor product. Even though the tensor product construction is a priori asymmetrical, both constructions produce isomorphic crystals, and in particular Sage’s crystals of tableaux are identical to Kashiwara’s.

**Note:** Using abstract crystals (i.e. they satisfy the axioms but do not arise from a representation of $U_q(\mathfrak{g})$), we can construct crystals $B, C$ such that $B \otimes C \neq C \otimes B$ (of course, using the same convention).

### Tensor products of crystals in Sage

You may construct the tensor product of several crystals in Sage using `crystals.TensorProduct`:

```python
sage: C = crystals.Letters("A2")
sage: T = crystals.TensorProduct(C,C,C); T
Full tensor product of the crystals [The crystal of letters for type ['A', 2], The crystal of letters for type ['A', 2], The crystal of letters for type ['A', 2]]
sage: T.cardinality()
27
sage: T.highest_weight_vectors()
([1, 1, 1], [2, 1, 1], [1, 2, 1], [3, 2, 1])
```

This crystal has four highest weight vectors. We may understand this as follows:

```python
sage: A2 = WeylCharacterRing("A2")
sage: C = crystals.Letters("A2")
sage: T = crystals.TensorProduct(C,C,C)
sage: chi_C = C.character(A2)
sage: chi_T = T.character(A2)
sage: chi_C
A2(1,0,0)
sage: chi_T
A2(1,1,1) + 2*A2(2,1,0) + A2(3,0,0)
sage: chi_T == chi_C^3
True
```

As expected, the character of $T$ is the cube of the character of $C$, and representations with highest weight $(1,1,1)$, $(3,0,0)$ and $(2,1,0)$. This decomposition is predicted by Frobenius-Schur duality: the multiplicity of $\pi^\text{GL(n)}_\lambda$ in $\otimes^3 C^3$ is the degree of $\pi^S^3_\lambda$.

It is useful to be able to select one irreducible constituent of $T$. If we only want one of the irreducible constituents of $T$, we can specify a list of highest weight vectors by the option `generators`. If the list has only one element, then we get an irreducible crystal. We can make four such crystals:

```python
sage: A2(1,1,1) + 2*A2(2,1,0) + A2(3,0,0)
sage: chi_T == chi_C^3
True
```
We see that two of these crystals are isomorphic, with character \(A2(2,1,0)\). Try:

```python
sage: A2 = WeylCharacterRing("A2")
sage: C = crystals.Letters("A2")
sage: T = crystals.TensorProduct(C,C,C)
sage: [T1,T2,T3,T4] = 
    [crystals.TensorProduct(C,C,C,generators=[v]) for v in T.highest_weight_vectors()]

sage: [B.cardinality() for B in [T1,T2,T3,T4]]
[10, 8, 8, 1]
sage: [B.character(A2) for B in [T1,T2,T3,T4]]
[A2(3,0,0), A2(2,1,0), A2(2,1,0), A2(1,1,1)]
```

Elements of `crystals.TensorProduct(A,B,C, ...)` are represented by sequences \([a,b,c, ...]\) with \(a\) in \(A\), \(b\) in \(B\), etc. This of course represents \(a \otimes b \otimes c \otimes \cdots\).

**Crystals of tableaux as tensor products of crystals**

Sage implements the `CrystalOfTableaux` as a subcrystal of a tensor product of the `ClassicalCrystalOfLetters`. You can see how its done as follows:

```python
sage: T = crystals.Tableaux("A4",shape=[3,2])
sage: v = T.highest_weight_vector().f(1).f(2).f(3).f(2).f(1).f(4).f(2).f(3); v
[[1, 2, 5], [3, 4]]
sage: list(v)
[3, 1, 4, 2, 5]
```

We’ve looked at the internal representation of \(v\), where it is represented as an element of the fourth tensor power of the `ClassicalCrystalOfLetters`. We see that the tableau:

```
  1 2 5
  3 4
```

is interpreted as the tensor:

```
 3 \otimes 1 \otimes 4 \otimes 2 \otimes 5
```

The elements of the tableau are read from bottom to top and from left to right. This is the *inverse middle-Eastern reading* of the tableau. See Hong and Kang, *loc. cit.* for discussion of the readings of a tableau.
Spin crystals

For the Cartan types $A_r$, $C_r$ or $G_2$, `CrystalOfTableaux` are capable of making any finite crystal. (For type $A_r$ it is necessary that the highest weight $\lambda$ be a partition.)

For Cartan types $B_r$ and $D_r$, there also exist spin representations. The corresponding crystals are implemented as *spin crystals*. For these types, `CrystalOfTableaux` also allows the input shape $\lambda$ to be half-integral if it is of height $r$. For example:

```sage
sage: C = crystals.Tableaux(['B',2], shape = [3/2, 1/2])
sage: C.list()

[[++, [[1]]], [++, [[2]]], [++, [[0]]], [++, [[-2]]], [++, [[-1]]], [++, [[-2]]], [++, [[-1]]], [+-, [[-1]]], [+-, [[-1]]], [+-, [[1]]], [+-, [[2]]], [+-, [[2]]], [+-, [[2]]], [+-, [[0]]], [+-, [[0]]], [+-, [[-2]]], [-+, [[-2]]], [-+, [[-1]]]]
```

Here the first list of $+$ and $-$ gives a spin column that is discussed in more detail in the next section and the second entry is a crystal of tableau element for $\lambda = (\lfloor \lambda_1 \rfloor, \lfloor \lambda_2 \rfloor, \ldots)$. For type $D_r$, we have the additional feature that there are two types of spin crystals. Hence in `CrystalOfTableaux` the $r$-th entry of $\lambda$ in this case can also take negative values:

```sage
sage: C = crystals.Tableaux(['D',3], shape = [1/2, 1/2,-1/2])
sage: C.list()

[[++, []], [+-, []], [++, []], [-+, []], [--, []]]
```

For rank two Cartan types, we also have `crystals.FastRankTwo` which gives a different fast implementation of these crystals:

```sage
sage: B = crystals.FastRankTwo(['B',2], shape=[3/2,1/2]); B
The fast crystal for B2 with shape [3/2,1/2]
sage: v = B.highest_weight_vector(); v.weight()
(3/2, 1/2)
```

Type B spin crystal

The spin crystal has highest weight $(1/2, \ldots, 1/2)$. This is the last fundamental weight. The irreducible representation with this weight is the spin representation of degree $2^r$. Its crystal is hand-coded in Sage:

```sage
sage: Cspin = crystals.Spins("B3"); Cspin
The crystal of spins for type ['B', 3]
sage: Cspin.cardinality()
8
```

The crystals with highest weight $\lambda$, where $\lambda$ is a half-integral weight, are constructed as a tensor product of a spin column and the highest weight crystal of the integer part of $\lambda$. For example, suppose that $\lambda = (3/2, 3/2, 1/2)$. The corresponding irreducible character will have degree $112$:

```sage
sage: B3 = WeylCharacterRing("B3")
sage: B3(3/2,3/2,1/2).degree()
112
```

So $B_\lambda$ will have $112$ elements. We can find it as a subcrystal of $Cspin \otimes B_\mu$, where $\mu = \lambda - (1/2, 1/2, 1/2) = (1, 1, 0)$:
We see that just taking the tensor product of these two crystals will produce a reducible crystal with three constituents, and we want to extract the one we want. We do that as follows:

```sage
c = C.cardinality()
112
```

Alternatively, we can get this directly from `CrystalOfTableaux`:

```sage
c = C.cardinality()
112
```

This is the desired crystal.

**Type D spin crystals**

A similar situation pertains for type $D_r$, but now there are two spin crystals, both of degree $2^r - 1$. These are hand-coded in sage:

```sage
[8, 8]
```

Similarly to type B crystal, we obtain crystal with spin weight by allowing for partitions with half-integer values, and the last entry can be negative depending on the type of the spin.

**Lusztig involution**

The Lusztig involution on a finite-dimensional highest weight crystal $B(\lambda)$ of highest weight $\lambda$ maps the highest weight vector to the lowest weight vector and the Kashiwara operator $f_i$ to $e_i^{*}$. For example, for type $A_n$ we have $i^* = n + 1 - i$, whereas for type $C_n$ we have $i^* = i$. For type $D_n$ and $n$ even also have $i^* = i$, but for $n$ odd this map interchanges nodes $n - 1$ and $n$. Here is how to achieve this in Sage:

```sage
[[2, 4], [3]]
```
For type $A_n$, the Lusztig involution is the same as the Schuetzenberger involution (which in Sage is defined on tableaux):

```python
sage: t = Tableau([[1,2],[3]])
sage: t.schuetzenberger_involution(n=4)
[[2, 4], [3]]
```

For all tableaux in a given crystal, this can be tested via:

```python
sage: B = crystals.Tableaux(['A',3],shape=[2])
sage: all(b.lusztig_involution().to_tableau() == b.to_tableau().schuetzenberger_involution(n=4) for b in B)
True
```

The Lusztig involution is also defined for finite-dimensional highest weight crystals of exceptional type:

```python
sage: C = CartanType(['E',6])
sage: La = C.root_system().weight_lattice().fundamental_weights()
sage: T = crystals.HighestWeight(La[1])
sage: t = T[4]; t
[(-2, 5)]
sage: t.lusztig_involution()
[(-3, 2)]
```

**Levi branching rules for crystals**

Let $G$ be a Lie group and $H$ a Levi subgroup. We have already seen that the Dynkin diagram of $H$ is obtained from that of $G$ by erasing one or more nodes.

If $C$ is a crystal for $G$, then we may obtain the corresponding crystal for $H$ by a similar process. For example if the Dynkin diagram for $H$ is obtained from the Dynkin diagram for $G$ by erasing the $i$-th node, then if we erase all the edges in the crystal $C$ that are labeled with $i$, we obtain a crystal for $H$.

In Sage this is achieved by specifying the index set used in the digraph method:

```python
sage: T = crystals.Tableaux(['D',4],shape=[1])
sage: G = T.digraph(index_set=[1,2,3])
```

We see that the type $D_4$ crystal indeed decomposes into two type $A_3$ components.
For more on branching rules, see *Maximal Subgroups and Branching Rules* or *Levi subgroups* for specifics on the Levi subgroups.

**Subcrystals**

Sometimes it might be desirable to work with a subcrystal of a crystal. For example, one might want to look at all \{2, 3, \ldots, n\} highest elements of a crystal and look at a particular such component:

```python
sage: T = crystals.Tableaux(['D',4],shape=[2,1])
sage: hw = [ t for t in T if t.is_highest_weight(index_set = [2,3,4]) ]; hw
[[[1, 1], [2]],
 [[1, 2], [2]],
 [[2, -1], [-2]],
 [[2, -1], [-1]],
 [[1, -1], [2]],
 [[2, -1], [3]],
 [[1, 2], [3]],
 [[2, 2], [3]],
 [[1, 2], [-2]],
 [[2, 2], [-2]],
 [[2, 2], [-1]]]
```

```python
sage: C = T.subcrystal(generators = [T(rows=[[2,-1],[3]])], index_set = [2,3,4])
sage: G = T.digraph(subset = C, index_set=[2,3,4])
```

**Affine Root System Basics**

Among infinite-dimensional Lie algebras, *Kac-Moody Lie algebras* are generalizations of finite-dimensional simple Lie algebras. They include finite-dimensional simple Lie algebras as special cases but are usually infinite-dimensional. Many concepts and results from the representation theory of finite-dimensional Lie groups and Lie algebras extend to Kac-Moody Lie algebras. This includes the root system, Weyl group, weight lattice, the parametrization of important representations (the integrable highest weight ones) by dominant weights and the Weyl character formula for these representations.

Among Kac-Moody Lie algebras, *affine Lie algebras* are an important infinite-dimensional class, and their infinite-dimensional integrable highest-weight representations are an important class among their representations. In this section many of the concepts are applicable to general Kac-Moody Lie algebras. However the code that we will discuss is mainly for the affine case. This is sufficiently different from the general case (and important) to merit special attention.

In this section we will review some of the Kac-Moody theory, taking [Kac] as our primary reference. We also recommend [KMPS]. This two volume set contains tables (in Volume 2) of quantities such as string functions and modular characteristics that you can now compute in Sage. Volume 1 contains an introduction to affine Lie algebras and their representations, including a valuable discussion of how they arise in string theory.

In this section and the next, we will also explain what facilities there are in Sage for computing with these. We will often restrict ourselves to the case of affine Lie algebras.
Cartan Matrix

See \[Kac\] Chapter 1 for this topic.

The basic data defining a Kac-Moody Lie algebra is a (generalized) Cartan matrix. This is a square matrix \( A = (a_{ij}) \) with diagonal entries equal to 2 and nonpositive off diagonal entries such that \( a_{ij} = 0 \) if and only if \( a_{ji} = 0 \). It is useful to assume that it is indecomposable and symmetrizable. Indecomposable means that it cannot be arranged into two diagonal blocks by permuting the rows and columns; and symmetrizable means that \( DA \) is symmetric for some invertible diagonal matrix \( D \).

Given a generalized Cartan matrix there is a vector space \( h \) containing vectors \( \alpha_i^\vee \) (called simple coroots) and vectors \( \alpha_i \in h^* \) (called simple roots) such that \( \langle \alpha_i^\vee, \alpha_j \rangle = \alpha_i^\vee(\alpha_j) = a_{ij} \). Moreover there exists a Kac-Moody Lie algebra \( g \) containing \( h \) as an abelian subalgebra that is generated by \( h \) and elements \( e_i \) and \( f_i \) such that

\[
[e_i, f_i] = \delta_{ij} \alpha_i^\vee, \quad [h, e_i] = \alpha_i(h)e_i, \quad [h, f_i] = -\alpha_i(h)f_i.
\]

(These conditions do not quite characterize \( g \), but they do if supplemented by the Serre relations, which we will not need or state.)

The significance of the diagonalizability assumption is that \( g \) admits an invariant symmetric bilinear form, and hence has a Casimir operator and a good representation theory.

The transpose of \( A \) is also a symmetrizable indecomposable generalized Cartan matrix, so there is a dual Cartan type in which the roots and coroots are interchanged.

In Sage, we may recover the Cartan matrix as follows:

```sage
sage: RootSystem(['B',2]).cartan_matrix()
[ 2 -1]
[-2  2]
sage: RootSystem(['B',2,1]).cartan_matrix()
[ 2  0 -1]
[ 0  2 -1]
[-2 -2  2]
```

If \( \det(A) = 0 \) and its nullspace is one-dimensional, then \( g \) is an affine Lie algebra, as in the second example above.

Untwisted Affine Kac-Moody Lie Algebras

One realization of affine Lie algebras, described in Chapter 7 of [Kac] begins with a finite-dimensional simple Lie algebra \( g^\circ \), with Cartan type \( X_\ell(1) \) in Sage). Tensoring with the Laurent polynomial ring gives the loop Lie algebra \( g^\circ \otimes \mathbb{C}[t, t^{-1}] \). This is the Lie algebra of vector fields in \( g^\circ \) on the circle. Then one may make a central extension:

\[
0 \rightarrow \mathbb{C} \cdot K \rightarrow g' \rightarrow g^\circ \otimes \mathbb{C}[t, t^{-1}] \rightarrow 0.
\]

After that, it is convenient to adjoin another basis element, which acts on \( g' \) as a derivation \( d \). If \( h^\circ \) is a Cartan subalgebra of \( g^\circ \), then we obtain a Cartan subalgebra \( h' \) of \( g' \) by adjoining the central element \( K \), and then a Cartan subalgebra \( h \) by further adjoining the derivation \( d \).

The resulting Lie algebra \( g \) is the untwisted affine Lie algebra. The Cartan type is designated to be \( X_{\ell}^{(1)} \) in Kac’ notation, which is rendered as \( ['X',1,1] \) or “X1~” in Sage. The Dynkin diagram of this Cartan type is the extended Dynkin-diagram of \( g^\circ \):
Thematic Tutorials, Release 9.7

From the Dynkin diagram, we can read off generators and relations for the affine Weyl group, which is a Coxeter group with generators $s_i$ of order 2, that commute if $i$ and $j$ are not adjacent in the Dynkin diagram, and otherwise are subject to a braid relation. We can infer some Levi subalgebras of $\mathfrak{g}$, obtained by omitting one node from the Dynkin diagram; particularly omitting the “affine node” 0 gives $E_6^\circ$, that is $\mathfrak{g}^\circ$.

The index set for the finite-dimensional Lie algebra $\mathfrak{g}^\circ$ is $I = \{1, 2, \ldots, \ell\}$. This means we label the roots, coroots etc. by $i \in I$. The index set for the affine Lie algebra $\mathfrak{g}$ adds one index 0 for the affine root $\alpha_0$.

The subset of $\lambda \in \mathfrak{h}^*$ characterized by $\lambda(\alpha_0^\vee) \in \mathbb{Z}$ for the coroots $\alpha_i^\vee$ is called the weight lattice $P$. There are two versions of the weight lattice, depending on whether we are working with $\mathfrak{g}$ or $\mathfrak{g}'$. The weight lattice of $\mathfrak{g}$ is called the extended weight lattice. We may create these as follows:

```sage
sage: RootSystem("A2~").weight_lattice()
Weight lattice of the Root system of type ['A', 2, 1]
sage: RootSystem("A2~").weight_lattice(extended=True)
Extended weight lattice of the Root system of type ['A', 2, 1]
```

Referring to the extended lattice, the term lattice is a slight misnomer because $P$ is not discrete; it contains all complex multiples of $\delta$, which is orthogonal to the coroots. However the image of $P$ in $\mathfrak{h}^*/C\delta$ is a bona fide lattice. Indeed, the fundamental weights are vectors $\Lambda_i \in \mathfrak{h}^*$ such that $\Lambda_i(\alpha_j^\vee) = \delta_{ij}$, and then

$$P = C\delta \oplus \bigoplus_{i=0}^\ell \mathbb{Z}\Lambda_i.$$

The Weyl vector $\rho$ is the sum of the fundamental weights. This plays the role as does the classical Weyl vector, which also equals half the sum of the positive roots, in the theory of finite semisimple Lie algebras. The weight lattice $P$ contains the root lattice $Q$, which is the lattice spanned by $\alpha_0, \alpha_1, \ldots, \alpha_\ell$.

Usually there is an advantage to working with $\mathfrak{g}$ instead of $\mathfrak{g}'$. (Thus we prefer the extended weight lattice, though this is not the default.) The reason for this is as follows. If $V$ is a representation of $\mathfrak{g}$ then usually the weight spaces $V_\lambda$, in a decomposition with respect to characters (weights) of $\mathfrak{h}$ are finite-dimensional; but the corresponding weight spaces for $\mathfrak{h}'$ would not be.

There are exceptions to this rule of preferring the extended weight lattice. In particular, we can construct non-trivial irreducible finite-dimensional representations of $\mathfrak{g}'$, and these cannot be lifted to $\mathfrak{g}$ (although they do have infinite-dimensional analogs). These certain finite-dimensional representations have crystal bases, which include Kirillov-Reshetikhin crystals. Thus, for Kirillov-Reshetikhin crystals we prefer to use the non-extended weight lattice. See Affine Finite Crystals.
Twisted Types

There are also twisted types with Cartan type $X^{(m)}_k$ or $[X',1,m]$ where $m$ is the order of an automorphism of the Dynkin diagram of $\mathfrak{g}$. These are described in [Kac] Chapter 8. Alternative descriptions of the twisted types may be found in [Macdonald2003]. Examining the tables Aff1, Aff2 and Aff3 in Chapter 4 of Kac, you will see that each twisted type is dual to an untwisted type (except $A^{(2)}_{2k}$). For example the twisted type $[E',6,2]$ (or $E^{(2)}_6$) in Aff2 is dual to the untwisted type $[F',4,1]$ (or $F^{(1)}_4$).

Referring to the above Dynkin diagram for $[E',6,1]$, if we collapse the nodes 1 and 6 together, and the nodes 3 and 5, we obtain the Dynkin diagram for $[E',6,2]$:

```
sage: CartanType(['E',6,2]).dynkin_diagram()
O---O---O=<=O---O
0 1 2 3 4
F4~*
```

We must explain why Sage calls this Cartan type $F4$~. The Cartan type $[F',4,1]$ is obtained by adding one Dynkin node to the Cartan type “F4”:

```
sage: CartanType(['F',4,1]).dynkin_diagram()
O---O---O=>=O---O
0 1 2 3 4
F4~
```

The Cartan types $[E',6,2]$ and $[F',4,1]$ (abbreviated F4~) are dual in the sense that long roots of one correspond to short roots of the other. (Thus $\alpha_0, \alpha_1$ and $\alpha_2$ are short roots of $[E',6,2]$, they are long roots of $[F',4,1]$.) More generally, every twisted affine type is dual to a unique untwisted type, and the Macdonald convention is to refer to the Cartan type as the dual of the corresponding untwisted type:

```
sage: CartanType(['F',4,1]).dual() == CartanType(['E',6,2])
True
```

Roots and Weights

A Kac-Moody Lie algebra $\mathfrak{g}$ has a triangular decomposition

$$\mathfrak{g} = \mathfrak{h} \oplus \mathfrak{n}_+ \oplus \mathfrak{n}_-$$

where $\mathfrak{n}_-$ and $\mathfrak{n}_+$ are locally nilpotent Lie algebras.

If $V$ is a $\mathfrak{g}$-module then we often have a weight space decomposition

$$V = \bigoplus_{\lambda \in \mathfrak{h}^*} V_\lambda$$

where $V_\lambda$ is finite-dimensional, and where $\mathfrak{h}$ acts by $X v = \lambda(X)v$ for $X \in \mathfrak{h}$, $v \in V_\lambda$. When $V_\lambda \neq 0$, the linear functional $\lambda$ is called a weight. The space $V_\lambda$ is called the weight space and its dimension is the multiplicity of the weight $\lambda$.

As a special case, $\mathfrak{g}$ is a module over itself under the adjoint representation, and it has a weight space decomposition.

The roots are the nonzero weights in the adjoint representation of $\mathfrak{g}$ on itself. In contrast with the finite-dimensional case, if $\mathfrak{g}$ is an infinite Kac-Moody Lie algebra there are two types of roots, called real and imaginary. The real roots have multiplicity 1, while the imaginary roots can have multiplicity $> 1$. In the case of the affine Kac-Moody Lie
algebra the imaginary roots have bounded multiplicity, while in non-affine cases the multiplicities of the imaginary roots is somewhat mysterious.

The roots may be divided into those in the adjoint representation of \( h \) on \( n_+ \), called **positive**, and those on \( n_- \), called **negative**.

Returning to the general module \( V \) with a weight space decomposition, a vector in the module \( V \) that is annihilated by \( n_+ \) is called a **highest weight vector**. If the space of highest weight vectors is one-dimensional, and if \( V \) is generated by a highest weight vector \( v \) then \( C v = V_\lambda \) for a weight \( \lambda \), called the highest weight, and \( V \) is called a highest weight representation.

If \( \lambda \) is any linear functional on \( h \), then there is a **universal highest weight module** \( M(\lambda) \) such that any highest weight module with highest weight \( \lambda \) is a quotient of \( M(\lambda) \). In particular \( M(\lambda) \) (which is also called a Verma module) has a unique irreducible quotient denoted \( L(\lambda) \). Looking ahead to crystal bases, the infinity crystal \( B(\infty) \) is a crystal base of the Verma module \( M(0) \).

A weight \( \lambda \in P \) is called **dominant** if

\[
\langle \lambda, \alpha_i^\vee \rangle = \lambda(\alpha_i^\vee) \geq 0
\]

for all simple coroots \( \alpha_i^\vee \). Let \( P^+ \) be the set of dominant weights.

A weight has a **level** which can be defined as its inner product with the canonical central element \( c \). Each weight knows its level:

```sage
sage: L = RootSystem(['E',6,1]).weight_lattice(extended=True)
sage: Lambda = L.fundamental_weights()
sage: [Lambda[i].level() for i in L.index_set()]
[1, 1, 2, 2, 3, 2, 1]
```

### Affine Root System and Weyl Group

We now specialize to affine Kac-Moody Lie algebras and their root systems. The basic reference for the affine root system and Weyl group is [Kac] Chapter 6.

In the untwisted affine case, the root system \( \Delta \) contains a copy of the root system \( \Delta^\circ \) of \( g^\circ \). The real roots consist of \( \alpha + n\delta \) with \( \alpha \in \Delta^\circ \) and \( n \in \mathbb{Z} \). The root is positive if either \( n = 0 \) and \( \alpha \in \Delta^\circ_+ \) or \( n > 0 \). The imaginary roots consist of \( n\delta \) with \( n \in \mathbb{Z} \) nonzero. See [Kac], Proposition 6.3 for a description of the root system in the twisted affine case.

The multiplicity \( m(\alpha) \) is the dimension of \( g_\alpha \). It is 1 if \( \alpha \) is a real root. For the untwisted affine Lie algebras, the multiplicity of an imaginary root is the rank \( \ell \) of \( g^\circ \). (For the twisted cases, see [Kac] Corollary 8.3.)

In most cases we recommend creating the weight lattice with the option \texttt{extended=True}:

```sage
sage: WL = RootSystem(['A',2,1]).weight_lattice(extended=True); WL
Extended weight lattice of the Root system of type ['A', 2, 1]
sage: WL.positive_roots()
Disjoint union of Family (Positive real roots of type ['A', 2, 1], Positive imaginary roots of type ['A', 2, 1])
sage: WL.simple_roots()
sage: WL.weyl_group()
Weyl Group of type ['A', 2, 1] (as a matrix group acting on the extended weight lattice)
sage: WL.basic_imaginary_roots()[0]
delta
```
Be aware that for the exceptional groups, the ordering of the indices are different from those in [Kac]. This is because Sage uses the Bourbaki ordering of the roots, and Kac does not. Thus in Bourbaki (and in Sage) the $G_2$ short root is $\alpha_1$:

```
sage: CartanType(['G',2,1]).dynkin_diagram()
3
0<=0---0
1 2 0
G2~
```

By contrast in Kac, $\alpha_2$ is the short root.

### Labels and Coxeter Number

Certain constants $a_i$ label the vertices $i = 0, \ldots, \ell$ in the tables Aff1, Aff2 and Aff3 in [Kac] Chapter 4. They are called *labels* by Kac and *marks* in [KMPS]. They play an important role in the theory. In Sage they are available as follows:

```
sage: CartanType(['B',5,1]).a()
Finite family {0: 1, 1: 1, 2: 2, 3: 2, 4: 2, 5: 2}
```

The column vector $a$ with these entries spans the nullspace of $A$:

```
sage: RS = RootSystem(['E',6,2]); RS
Root system of type ['E', 6, 1]^*

sage: A = RS.cartan_matrix(); A
[ 2 -1  0  0  0]
[-1  2 -1  0  0]
[ 0 -1  2  0  0]
[ 0  0 -1  2 -1]
[ 0  0  0 -1  2]

sage: ann = Matrix([[v] for v in RS.cartan_type().a()]); ann
[1]
[2]
[3]
[2]
[1]

sage: A*ann
[0]
[0]
[0]
[0]
[0]
```

The nullroot $\delta = \sum_{i \in I} a_i \alpha_i$:

```
sage: WL = RootSystem('C3~').weight_lattice(extended=True); WL
Extended weight lattice of the Root system of type ['C', 3, 1]

sage: sum(WL.cartan_type().a()[i]*WL.simple_root(i) for i in WL.cartan_type().index_set())
delta
```

The number $h = \sum_{i \in I} a_i$ is called the *Coxeter number*. In the untwisted case it is the order of a Coxeter element of the finite Weyl group of $g^\circ$. The *dual Coxeter number* $h^\vee$ is the Coxeter number of the dual root system. It appears frequently in representation theory. The Coxeter number and dual Coxeter number may be computed as follow:
The Weyl Group

The ambient space of the root system comes with an (indefinite) inner product. The real roots have nonzero length but the imaginary roots are isotropic. If \( \alpha \) is a real root we may define a reflection \( r_\alpha \) in the hyperplane orthogonal to \( \alpha \). In particular the \( \ell + 1 \) reflections \( s_i \) with respect to the simple positive roots \( \alpha_i \) \( (i = 0, 1, 2, \ldots, \ell) \) generate a Coxeter group. This is the Weyl group \( W \).

Illustrating how to use Sage to compute the action of \( W \) on the weight lattice:

```sage
sage: L = RootSystem("A2~").weight_lattice(extended=True)
sage: Lambda = L.fundamental_weights()
sage: delta = L.null_root()
sage: W = L.weyl_group(prefix="s")
sage: s0,s1,s2 = W.simple_reflections()
sage: [(s0*s1*s2*s1).action(x) - x for x in Lambda]
[-2*Lambda[0] + Lambda[1] + Lambda[2] - delta, 
sage: [s0.action(x) for x in Lambda]
sage: s0.action(delta)
delta
```

The extended affine Weyl Group

The subgroup \( W^0 \) generated by \( s_1, \ldots, s_\ell \) is a finite Coxeter group that may be identified with the Weyl group of the finite-dimensional simple Lie algebra \( g^0 \).

Geometrically, \( W \) may be interpreted as the semidirect product of the finite Weyl group \( W^0 \) by a discrete group of translations \( Q^\vee \) isomorphic to the coroot lattice. A larger extended affine Weyl group is the semidirect product of \( W^0 \) by the coweight lattice \( P^\vee \). If \( P^\vee \) is strictly larger than \( Q^\vee \) this is not a Coxeter group but arises naturally in many problems. It may be constructed in Sage as follows:

```sage
sage: E = ExtendedAffineWeylGroup(["A",2,1]); E
Extended affine Weyl group of type ['A', 2, 1]
```

See the documentation in extended_affine_weyl_group if you need this.
Integrable Highest Weight Representations of Affine Lie algebras

In this section \( \mathfrak{g} \) can be an arbitrary Kac-Moody Lie Algebra made with a symmetrizable, indecomposable Cartan matrix.

Suppose that \( V \) is a representation with a weight space decomposition as in Roots and Weights. Let \( \alpha \) be a real root, and let \( \mathfrak{g}_\alpha \) be the corresponding root space, that is, the one-dimensional weight space for \( \alpha \) in the adjoint representation of \( \mathfrak{g} \) on itself. Then \( -\alpha \) is also a root. The two one-dimensional spaces \( \mathfrak{g}_\alpha \) and \( \mathfrak{g}_{-\alpha} \) generate a Lie algebra isomorphic to \( \mathfrak{sl}_2 \). The module \( V \) is called integrable if for each such \( \alpha \) the representation of \( \mathfrak{sl}_2 \) obtained this way integrates to a representation of the Lie group \( \text{SL}_2 \). Since this group contains an element that stabilizes \( \mathfrak{h} \) and induces the corresponding simple reflection on the weight lattice, integrability implies that the weight multiplicities are invariant under the action of the Weyl group.

If the Kac-Moody Lie algebra \( \mathfrak{g} \) is finite-dimensional then the integrable highest weight representations are just the irreducible finite-dimensional ones. For a general Kac-Moody Lie algebra the integrable highest weight representations are the analogs of the finite-dimensional ones, that is, with \texttt{WeylCharacterRing} elements. Their theory has many aspects in common with the finite-dimensional representations of finite-dimensional simple Lie algebras, such as the parametrization by dominant weights, and generalizations of the Weyl denominator and character formulas, due to Macdonald and Kac respectively.

If \( \Lambda \) is a dominant weight, then the irreducible highest weight module \( L(\Lambda) \) defined in Roots and Weights is integrable. Moreover every highest weight integrable representation arises this way, so these representations are in bijection with the cone \( P^+ \) of dominant weights.

The affine case

Now we assume that \( \mathfrak{g} \) is affine. The integrable highest weight representations (and their crystals) are extremely interesting. Integrable highest weight representations of \( \mathfrak{g} \) arise in a variety of contexts, from string theory to the modular representation theory of the symmetric groups, and the theory of modular forms. The representation \( L(\Lambda_0) \) is particularly ubiquitous and is called the basic representation.

Therefore in [KMPS] (published in 1990) tabulated data for many of these representations. They wrote:

We present a vast quantity of numerical data in tabular form, this being the only source for such information.

The computations are tedious and not particularly straightforward when it is necessary to carry them out individually. We hope the appearance of this book will spur interest in a field that has become, in barely 20 years, deeply rewarding and full of promise for the future. It would indeed be gratifying if these tables were to appear to the scientists of 2040 as obsolete as the dust-gathering compilations of transcendental functions appear for us today because of their availability on every pocket calculator.

As we will explain, Sage can reproduce the contents of these tables. Moreover the tables in [KMPS] are limited to the untwisted types, but Sage also implements the twisted types.

Although Sage can reproduce the tables in the second volume of [KMPS], the work remains very useful. The first volume is a down-to-earth and very helpful exposition of the theory of integrable representations of affine Lie algebras with explicit examples and explanations of the connections with mathematical physics and vertex operators.
The support of an integrable highest weight representation

Let \( \Lambda \in P^+ \) and let \( V = L(\Lambda) \) be the integrable representation with highest weight \( \Lambda \). If \( \mu \) is another weight, let \( \text{mult}(\mu) \) denote the multiplicity of the weight \( \mu \) in \( L(\Lambda) \). Define the support of the representation \( \text{supp}(V) \) to be the set of \( \mu \) such that \( \text{mult}(\mu) > 0 \).

If \( \text{mult}(\mu) > 0 \) then \( \lambda - \mu \) is a linear combination of the simple roots with nonnegative integer coefficients. Moreover \( \text{supp}(V) \) is contained in the paraboloid

\[
(\Lambda + \rho|\Lambda + \rho) - (\mu + \rho|\mu + \rho) \geq 0
\]

where \( (\ |\ ) \) is the invariant inner product on the weight lattice and \( \rho \) is the Weyl vector (Untwisted Affine Kac-Moody Lie Algebras). Moreover if \( \mu \in \text{supp}(V) \) then \( \Lambda - \mu \) is an element of the root lattice \( Q \) ([Kac], Propositions 11.3 and 11.4).

We organize the weight multiplicities into sequences called string functions or strings as follows. By [Kac], Proposition 11.3 or Corollary 11.9, for fixed \( \mu \) the function \( \text{mult}(\mu - k\delta) \) of \( k \) is an increasing sequence. We adjust \( \mu \) by a multiple of \( \delta \) to the beginning of the positive part of the sequence. Thus we define \( \mu \) to be maximal if \( \text{mult}(\mu) \neq 0 \) but \( \text{mult}(\mu + \delta) = 0 \).

Since \( \delta \) is fixed under the action of the affine Weyl group, and since the weight multiplicities are Weyl group invariant, the function \( k \mapsto \text{mult}(\mu - k\delta) \) is unchanged if \( \mu \) is replaced by \( w(\mu) \) for some Weyl group element \( w \). Now every Weyl orbit contains a dominant weight. Therefore in enumerating the string we may assume that the weight \( \mu \) is dominant.

There are only a finite number of dominant maximal weights. Thus there are only a finite number of such strings to be computed.

Modular Forms

Remarkably, [KacPeterson] showed that each string is the set of Fourier coefficients of a weakly holomorphic modular form; see also [Kac] Chapters 12 and 13. Here weakly holomorphic modular means that the form is allowed to have poles at cusps.

To this end we define the modular characteristic

\[
m_{\Lambda} = \frac{|\Lambda + \rho|^2}{2(k + h^\vee)} - \frac{\rho^2}{2h^\vee}.
\]

Here \( k = (\Lambda|\delta) \) is the level of the representation and \( h^\vee \) is the dual Coxeter number (Labels and Coxeter Number). If \( \mu \) is a weight, define

\[
m_{\Lambda,\mu} = m_{\Lambda} - \frac{\mu^2}{2k}.
\]

Let \( \Lambda \) be a weight, which we may assume maximal. Then Kac and Peterson defined the string function

\[
c^\Lambda_\mu = q^{m_{\Lambda,\mu}} \sum_{n \in \mathbb{Z}} \text{mult}(\mu - n\delta)q^n.
\]

Although these do arise as partition functions in string theory, the term “string” here does not refer to physical strings.

The string function \( c^\Lambda_\mu \) is a weakly holomorphic modular form, possibly of half-integral weight. See [Kac], Corollary 13.10, or [KacPeterson]. It can have poles at infinity, but multiplying \( c^\Lambda_\mu \) by \( \eta(\tau)^{\dim g^\circ} \) gives a holomorphic modular form (for some level). Here \( \eta \) is the Dedekind eta function:

\[
\eta(\tau) = q^{1/24} \prod_{k=1}^{\infty} (1 - q^k), \quad q = e^{2\pi i \tau}.
\]

The weight of this modular form \( \eta(\tau)^{\dim g^\circ} c^\Lambda_\mu \) is the number of positive roots of \( g^\circ \).
Sage methods for integrable representations

In this section we will show how to use Sage to compute with integrable highest weight representations of affine Lie algebras. For further documentation, see the reference manual `IntegrableRepresentation`.

In the following example, we work with the integrable representation with highest weight $2\Lambda_0$ for $\widehat{\mathfrak{sl}}_2$, that is, $A_1^{(1)}$. First we create a dominant weight in the extended weight lattice, then create the `IntegrableRepresentation` class. We compute the strings. There are two, since there are two dominant maximal weights. One of them is the highest weight $2\Lambda_0$, and the other is $2\Lambda_1 - \delta$:

```python
sage: L = RootSystem("A1~").weight_lattice(extended=True)
sage: Lambda = L.fundamental_weights()
sage: delta = L.null_root()
sage: W = L.weyl_group(prefix="s")
sage: s0, s1 = W.simple_reflections()
sage: V = IntegrableRepresentation(2*Lambda[0])
sage: V.strings()
{2*Lambda[0]: [1, 1, 3, 5, 10, 16, 28, 43, 70, 105, 161, 236],
  2*Lambda[1] - delta: [1, 2, 4, 7, 13, 21, 35, 55, 86, 130, 196, 287]}
sage: mw1, mw2 = V.dominant_maximal_weights(); mw1, mw2
(2*Lambda[0], 2*Lambda[1] - delta)
```

We see there are two dominant maximal weights, $2\Lambda_0$ and $2\Lambda_1 - \delta$. We obtain every maximal weight from these by applying Weyl group elements. These lie inside the paraboloid described in *The support of an integrable highest weight representation*. Here are a few more maximal weights:

```python
sage: pairs = [(s0*s1*s0, mw1), (s0*s1, mw2), (s0, mw1), (W.one(), mw2),
            ....: (W.one(), mw1), (s1, mw2), (s1*s0, mw1), (s1*s0*s1, mw2)]
sage: [w.action(mw) for (w, mw) in pairs]
[-6*Lambda[0] + 8*Lambda[1] - 8*delta,
 -4*Lambda[0] + 6*Lambda[1] - 5*delta,
 -2*Lambda[0] + 4*Lambda[1] - 2*delta,
  2*Lambda[1] - delta,
  2*Lambda[0],
  4*Lambda[0] - 2*Lambda[1] - delta,
  6*Lambda[0] - 4*Lambda[1] - 2*delta,
  8*Lambda[0] - 6*Lambda[1] - 5*delta]
```

We confirm that the string function for one in the Weyl orbit is the same as that for $mw2$, calculated above:

```python
sage: s1.action(mw2)
4*Lambda[0] - 2*Lambda[1] - delta
sage: [V.mult(s0.action(mw2)-k*delta) for k in [0..10]]
[1, 2, 4, 7, 13, 21, 35, 55, 86, 130, 196]
```

String functions of integrable representations often appear in the Online Encyclopedia of Integer Sequences:

```python
sage: [oeis(x) for x in V.strings().values()]  # optional - internet
[0: A233758: Bisection of A006950 (the even part),
  0: A233759: Bisection of A006950 (the odd part).]
```

Reading what the OEIS tells us about the sequence OEIS sequence A006950, we learn that the two strings are the odd
and even parts of the series
\[
\prod_{k=1}^{\infty} \frac{1 + q^{2k-1}}{1 - q^{2k}} = \prod_{k=1}^{\infty} \frac{1 - q^{2k}}{(1 - q^{2k})(1 - q^{4k})} = q^{1/8} \frac{\eta(2\tau)}{\eta(\tau)\eta(4\tau)}
\]
This is not a modular form because of the factor \(q^{1/8}\) in front of the ratio of eta functions.

Let us confirm what the Online Encyclopedia tells us by computing the above product:

```
 sage: PS.<q> = PowerSeriesRing(QQ)
 sage: prod([(1+q^(2*k-1))/(1-q^(2*k)) for k in [1..20]])
 1 + q + q^2 + 2*q^3 + 3*q^4 + 4*q^5 + 5*q^6 + 7*q^7 + 10*q^8 + 13*q^9 + 16*q^10 + 21*q^11 + 28*q^12 + 35*q^13 + 43*q^14 + 55*q^15 + 70*q^16 + 86*q^17 + 105*q^18 + 130*q^19 + O(q^20)
```

We see the values of the two strings interspersed in this product, with the \(2\Lambda_0\) string values in the even positions and the \(2\Lambda_1 - \delta\) values in the odd positions.

To compute \(c_{\Lambda_0}^{\Lambda_0}\), which is guaranteed to be a modular form, we must compute the modular characteristics. We are interested in the cases where \(\lambda\) is one of the two dominant maximal weights:

```
 sage: [V.modular_characteristic(x) for x in [2*Lambda[0], 2*Lambda[1]-delta]]
 [-1/16, 7/16]
```

This gives us the string functions
\[
c_{\Lambda_0}^{\Lambda_0} = q^{-1/16}(1 + q + 3q^2 + 5q^3 + 10q^4 + 16q^5 + \cdots),
\]
\[
c_{\Lambda_0}^{\Lambda_1-\delta} = q^{7/16}(1 + 2q + 4q^2 + 7q^3 + 13q^4 + 21q^5 + \cdots).
\]

These are both weakly holomorphic modular forms. Any linear combination of these two is also a weakly holomorphic modular form. For example we may replace \(\tau\) by \(\tau/2\) in our previous identity and get
\[
c_{\Lambda_0}^{\Lambda_0} + c_{\Lambda_0}^{\Lambda_1-\delta} = \frac{\eta(\tau)}{\eta(\tau/2)\eta(2\tau)}.
\]

Many more examples may be found in [KacPeterson] and [KMPS].

Let \(V\) be the integrable highest weight representation with highest weight \(\Lambda\). If \(\mu\) is in the support of \(V\) then \(\Lambda - \mu\) is of the form \(\sum n_i\alpha_i\) where \(\alpha_i\) are the simple roots. Sage employs an internal representation of the weights as tuples \((n_0, n_1, \ldots)\). You can convert weights to and from this notation as follows:

```
 sage: L = RootSystem(['E',6,2]).weight_lattice(extended=True)
 sage: Lambda = L.fundamental_weights()
 sage: delta = L.null_root()
 sage: V = IntegrableRepresentation(Lambda[0])
 sage: V.strings()
 {Lambda[0]: [1, 2, 7, 14, 35, 66, 140, 252, 485, 840, 1512, 2534]}
 sage: V.to_weight((1,2,0,1,0))
 (1, 2, 0, 1, 0)
 sage: V.from_weight(Lambda[0]-delta)
 (1, 2, 3, 2, 1)
```

In reporting the strings, one may set the optional parameter depth to get more or fewer values. In certain cases even the first coefficient of the string is significant. See [JayneMisra2014] and [KimLeeOh2017].

Catalan numbers (OEIS sequence A000108):
sage: P = RootSystem(['A',12,1]).weight_lattice(extended=true)
sage: Lambda = P.fundamental_weights()
sage: IntegrableRepresentation(2*Lambda[0]).strings(depth=1) # long time
{2*Lambda[0]: [1],
 Lambda[1] + Lambda[12] - delta: [1],
 Lambda[3] + Lambda[10] - 3*delta: [5],
 Lambda[5] + Lambda[8] - 5*delta: [42],

Catalan triangle numbers (OEIS sequence A000245):
sage: sorted(IntegrableRepresentation(Lambda[0]+Lambda[2]).strings(depth=1).values()) # long time
[[1], [3], [9], [12], [28], [90], [297]]

Central binomial coefficients (OEIS sequence A001700, OEIS sequence A128015):
sage: P = RootSystem(['B',8,1]).weight_lattice(extended=true)
sage: Lambda = P.fundamental_weights()
sage: sorted(IntegrableRepresentation(Lambda[0]+Lambda[1]).strings(depth=1).values()) # long time
[[1], [1], [1], [3], [3], [10], [10], [35], [35], [126]]

Affine Finite Crystals

In this document we briefly explain the construction and implementation of the Kirillov–Reshetikhin crystals of [FourierEtAl2009].

Kirillov–Reshetikhin (KR) crystals are finite-dimensional affine crystals corresponding to Kirillov–Reshetikhin modules. They were first conjectured to exist in [HatayamaEtAl2001]. The proof of their existence for nonexceptional types was given in [OkadoSchilling2008] and their combinatorial models were constructed in [FourierEtAl2009]. Kirillov–Reshetikhin crystals $B^{r,s}_n$ are indexed first by their type (like $A^{(1)}_n$, $B^{(1)}_n$, ...) with underlying index set $I = \{0, 1, \ldots, n\}$ and two integers $r$ and $s$. The integers $s$ only needs to satisfy $s > 0$, whereas $r$ is a node of the finite Dynkin diagram $r \in I \setminus \{0\}$.

Their construction relies on several cases which we discuss separately. In all cases when removing the zero arrows, the crystal decomposes as a (direct sum of) classical crystals which gives the crystal structure for the index set $I_0 = \{1, 2, \ldots, n\}$. Then the zero arrows are added by either exploiting a symmetry of the Dynkin diagram or by using embeddings of crystals.

Type $A^{(1)}_n$

The Dynkin diagram for affine type $A$ has a rotational symmetry mapping $\sigma : i \mapsto i + 1$ where we view the indices modulo $n + 1$:

sage: C = CartanType(['A',3,1])
sage: C.dynkin_diagram()
0
0--------+
|   |
(continues on next page)
The classical decomposition of $B^{r,s}$ is the $A_n$ highest weight crystal $B(s\omega_r)$ or equivalently the crystal of tableaux labelled by the rectangular partition $(s^r)$:

$$B^{r,s} \cong B(s\omega_r) \quad \text{as a } \{1, 2, \ldots, n\}-\text{crystal}$$

In Sage we can see this via:

```sage
K = crystals.KirillovReshetikhin(['A',3,1],1,1)
sage: K.classical_decomposition()
```

The crystal of tableaux of type ['A', 3] and shape(s) [[1]]

```sage
K.list()
```

[[[1]], [[2]], [[3]], [[4]]]

```sage
K = crystals.KirillovReshetikhin(['A',3,1],2,1)
sage: K.classical_decomposition()
The crystal of tableaux of type ['A', 3] and shape(s) [[1, 1]]
```

One can change between the classical and affine crystal using the methods lift and retract:

```sage
b = K(rows=[[1], [3]]); type(b)
```

<class 'sage.combinat.crystals.kirillov_reshetikhin.KR_type_A_with_category.element_class'>

```sage
b.lift()
```

[[1], [3]]

```sage
type(b.lift())
```

<class 'sage.combinat.crystals.tensor_product.CrystalOfTableaux_with_category.element_class'>

```sage
b = crystals.Tableaux(['A',3], shape = [1,1])(rows=[[1], [3]])
sage: K.retract(b)
```

[[1], [3]]

```sage
type(K.retract(b))
```

<class 'sage.combinat.crystals.kirillov_reshetikhin.KR_type_A_with_category.element_class'>

The 0-arrows are obtained using the analogue of $\sigma$, called the promotion operator $pr$, on the level of crystals via:

$$f_0 = pr^{-1} \circ f_1 \circ pr$$

$$e_0 = pr^{-1} \circ e_1 \circ pr$$

In Sage this can be achieved as follows:

```sage
K = crystals.KirillovReshetikhin(['A',3,1],2,1)
sage: b = K.module_generator(); b
```

[[1], [2]]

```sage
b.f(0)
```

```sage
b.e(0)
```

(continues on next page)
KR crystals are level $0$ crystals, meaning that the weight of all elements in these crystals is zero:

```python
sage: K = crystals.KirillovReshetikhin(['A',3,1],2,1)
sage: b = K.module_generator(); b.weight()
-Lambda[0] + Lambda[2]
sage: b.weight().level()
0
```

The KR crystal $B_{1,1}^{1,1}$ of type $A_2^{(1)}$ looks as follows:

![KR crystal diagram]

In Sage this can be obtained via:

```python
sage: K = crystals.KirillovReshetikhin(['A',2,1],1,1)
sage: G = K.digraph()
sage: view(G, tightpage=True) # optional - dot2tex graphviz, not tested (opens external_window)
```

### Types $D_n^{(1)}$, $B_n^{(1)}$, $A_{2n-1}^{(2)}$

The Dynkin diagrams for types $D_n^{(1)}$, $B_n^{(1)}$, $A_{2n-1}^{(2)}$ are invariant under interchanging nodes 0 and 1:

```python
sage: n = 5
sage: C = CartanType(['D',n,1]); C.dynkin_diagram()
```

```
D5~
```

```
0 0
| |
0---0---0---0
1 2 3 4
```

```
sage: C = CartanType(['B',n,1]); C.dynkin_diagram()
```

```
0 0
| |
```

(continues on next page)
The underlying classical algebras obtained when removing node 0 are type $g_0 = D_n, B_n, C_n$, respectively. The classical decomposition into a $g_0$ crystal is a direct sum:

$$B'^{rs} \cong \bigoplus_{\lambda} B(\lambda)$$

as a \{1, 2, …, n\}-crystal

where $\lambda$ is obtained from $s\omega_r$ (or equivalently a rectangular partition of shape $(s')$) by removing vertical dominoes. This in fact only holds in the ranges $1 \leq r \leq n - 2$ for type $D_n^{(1)}$, and $1 \leq r \leq n$ for types $B_n^{(1)}$ and $A_{2n-1}^{(2)}$:

For type $B_n^{(1)}$ and $r = n$, one needs to be aware that $\omega_n$ is a spin weight and hence corresponds in the partition language to a column of height $n$ and width $1/2$:

As for type $A_n^{(1)}$, the Dynkin automorphism induces a promotion-type operator $\sigma$ on the level of crystals. In this case in however happen that the automorphism changes between classical components:

This operator $\sigma$ is used to define the affine crystal operators:

$$f_0 = \sigma \circ f_1 \circ \sigma$$

$$e_0 = \sigma \circ e_1 \circ \sigma$$

The KR crystals $B_1^{1-1}$ of types $D_3^{(1)}$, $B_2^{(1)}$, and $A_5^{(2)}$ are, respectively:
Type $C_n^{(1)}$

The Dynkin diagram of type $C_n^{(1)}$ has a symmetry $\sigma(i) = n - i$:

\[
\begin{array}{cccccc}
0 & \longrightarrow & 1 & \longrightarrow & 2 & \longrightarrow & 3 & \longrightarrow & 4 & \longrightarrow & 0
\end{array}
\]

The classical subalgebra when removing the 0 node is of type $C_n$.

However, in this case the crystal $B^{r,s}$ is not constructed using $\sigma$, but rather using a virtual crystal construction. $B^{r,s}$ of type $C_n^{(1)}$ is realized inside $\hat{V}^{r,s}$ of type $A_{2n+1}^{(2)}$ using:

\[
e_0 = \hat{e}_0 \hat{e}_1 \quad \text{and} \quad e_i = \hat{e}_{i+1} \quad \text{for} \quad 1 \leq i \leq n
\]

\[
f_0 = \hat{f}_0 \hat{f}_1 \quad \text{and} \quad f_i = \hat{f}_{i+1} \quad \text{for} \quad 1 \leq i \leq n
\]

where $\hat{e}_i$ and $\hat{f}_i$ are the crystal operator in the ambient crystal $\hat{V}^{r,s}$:

\[
sage: K = crystals.KirillovReshetikhin(['C',3,1],1,2); K.ambient_crystal()
sage: K.classical_decomposition()
\]

The classical decomposition for $1 \leq r < n$ is given by:

\[
B^{r,s} \cong \bigoplus_{\lambda} B(\lambda) \quad \text{as a} \{1,2,\ldots,n\}-\text{crystal}
\]

where $\lambda$ is obtained from $s\omega_r$ (or equivalently a rectangular partition of shape $(s^r)$) by removing horizontal dominoes:

\[
sage: K = crystals.KirillovReshetikhin(['C',3,1],2,4)
sage: K.classical_decomposition()
\]

The KR crystal $B^{1,1}$ of type $C_2^{(1)}$ looks as follows:
Types $D_{n+1}^{(2)}, A_{2n}^{(2)}$

The Dynkin diagrams of types $D_{n+1}^{(2)}$ and $A_{2n}^{(2)}$ look as follows:

\[\text{sage: } \text{C = CartanType(['D',5,2]); C.dynkin_diagram()}
\]
\[0=<=O---O---O=>=O
\]
\[0 \quad 1 \quad 2 \quad 3 \quad 4\]
\[C4~\]

\[\text{sage: } \text{C = CartanType(['A',8,2]); C.dynkin_diagram()}
\]
\[O=<=O---O---O=<=O
\]
\[0 \quad 1 \quad 2 \quad 3 \quad 4\]
\[BC4~\]

The classical subdiagram is of type $B_n$ for type $D_{n+1}^{(2)}$ and of type $C_n$ for type $A_{2n}^{(2)}$. The classical decomposition for these KR crystals for $1 \leq r < n$ for type $D_{n+1}^{(2)}$ and $1 \leq r \leq n$ for type $A_{2n}^{(2)}$ is given by:

\[B^{r,s} \cong \bigoplus_{\lambda} B(\lambda)\]

as a \{1, 2, \ldots, n\}-crystal

where $\lambda$ is obtained from $s\omega_r$ (or equivalently a rectangular partition of shape $(s^r)$) by removing single boxes:

\[\text{sage: } \text{K = crystals.KirillovReshetikhin(['D',5,2],2,2)}
\]
\[\text{sage: } \text{K.classical_decomposition()}
\]
The crystal of tableaux of type ['B', 4] and shape(s) [[], [1], [2], [1, 1], [2, 1], [2, \ldots, 2]]

\[\text{sage: } \text{K = crystals.KirillovReshetikhin(['A',8,2],2,2)}
\]
\[\text{sage: } \text{K.classical_decomposition()}
\]
The crystal of tableaux of type ['C', 4] and shape(s) [[], [1], [2], [1, 1], [2, 1], [2, \ldots, 2]]

The KR crystals are constructed using an injective map into a KR crystal of type $C_n^{(1)}$

\[S : B^{r,s} \to B^{r,2s}_{C_n^{(1)}}\]

such that $S(e_i b) = e_i^{m_i} S(b)$ and $S(f_i b) = f_i^{m_i} S(b)$

where

\[(m_0, \ldots, m_n) = (1, 2, \ldots, 2, 1)\]

for type $D_{n+1}^{(2)}$ and

\[(1, 2, \ldots, 2, 2)\]

for type $A_{2n}^{(2)}$.

\[\text{sage: } \text{K = crystals.KirillovReshetikhin(['D',5,2],1,2); K.ambient_crystal()}
\]
Kirillov-Reshetikhin crystal of type ['C', 4, 1] with (r,s)=(1,4)

\[\text{sage: } \text{K = crystals.KirillovReshetikhin(['A',8,2],1,2); K.ambient_crystal()}
\]
Kirillov-Reshetikhin crystal of type ['C', 4, 1] with (r,s)=(1,4)

The KR crystals $B_{1,1}^{1,1}$ of type $D_{3}^{(2)}$ and $A_{4}^{(2)}$ look as follows:
As you can see from the Dynkin diagram for type $A_{2n}^{(2)}$, mapping the nodes $i \mapsto n - i$ yields the same diagram, but with relabelled nodes. In this case the classical subdiagram is of type $B_n$ instead of $C_n$. One can also construct the KR crystal $B^{r-s}$ of type $A_{2n}^{(2)}$ based on this classical decomposition. In this case the classical decomposition is the sum over all weights obtained from $s\omega_r$ by removing horizontal dominoes:

```
sage: C = CartanType(['A',6,2]).dual()
sage: Kdual = crystals.KirillovReshetikhin(C,2,2)
sage: Kdual.classical_decomposition()
The crystal of tableaux of type ['B', 3] and shape(s) [[], [2], [2, 2]]
```

Looking at the picture, one can see that this implementation is isomorphic to the other implementation based on the $C_n$ decomposition up to a relabeling of the arrows:

```
sage: C = CartanType(['A',4,2])
sage: K = crystals.KirillovReshetikhin(C,1,1)
sage: Kdual = crystals.KirillovReshetikhin(C.dual(),1,1)
sage: G = K.digraph()
sage: Gdual = Kdual.digraph()
sage: f = { 1:1, 0:2, 2:0 }
sage: for u,v,label in Gdual.edges(sort=False):
    ....:     Gdual.set_edge_label(u,v,f[label])
sage: G.is_isomorphic(Gdual, edge_labels = True)
True
```
Exceptional nodes

The KR crystals $B^{n,s}$ for types $C_n^{(1)}$ and $D_{n+1}^{(2)}$ were excluded from the above discussion. They are associated to the exceptional node $r = n$ and in this case the classical decomposition is irreducible:

$$B^{n,s} \cong B(s \omega_n).$$

In Sage:

```python
sage: K = crystals.KirillovReshetikhin(['C',2,1],2,1)
sage: K.classical_decomposition()
The crystal of tableaux of type ['C', 2] and shape(s) [[1, 1]]
```

```python
sage: K = crystals.KirillovReshetikhin(['D',3,2],2,1)
sage: K.classical_decomposition()
The crystal of tableaux of type ['B', 2] and shape(s) [[1/2, 1/2]]
```
The KR crystals $B^{n,s}$ and $B^{n-1,s}$ of type $D_n^{(1)}$ are also special. They decompose as:

$$B^{n,s} \cong B(s\omega_n) \quad \text{and} \quad B^{n-1,s} \cong B(s\omega_{n-1}).$$

```sage
sage: K = crystals.KirillovReshetikhin(['D',4,1],4,1)
sage: K.classical_decomposition()
The crystal of tableaux of type ['D', 4] and shape(s) [[1/2, 1/2, 1/2, 1/2]]
sage: K = crystals.KirillovReshetikhin(['D',4,1],3,1)
sage: K.classical_decomposition()
The crystal of tableaux of type ['D', 4] and shape(s) [[1/2, 1/2, 1/2, -1/2]]
```

**Type $E_6^{(1)}$**

In [JonesEtAl2010] the KR crystals $B^{r,s}$ for $r = 1,2,6$ in type $E_6^{(1)}$ were constructed exploiting again a Dynkin diagram automorphism, namely the automorphism $\sigma$ of order 3 which maps $0 \mapsto 1 \mapsto 6 \mapsto 0$:

```sage
sage: C = CartanType(['E',6,1]); C.dynkin_diagram()
 0  2
 |  |
 0---0---0---0---0---0
1  3  4  5  6

E6~
```

The crystals $B^{1,s}$ and $B^{6,s}$ are irreducible as classical crystals:

```sage
sage: K = crystals.KirillovReshetikhin(['E',6,1],1,1)
sage: K.classical_decomposition()
Direct sum of the crystals Family (Finite dimensional highest weight crystal of type ['E', 6], highest weight Lambda[1],)
sage: K = crystals.KirillovReshetikhin(['E',6,1],6,1)
sage: K.classical_decomposition()
Direct sum of the crystals Family (Finite dimensional highest weight crystal of type ['E', 6], highest weight Lambda[6],)
```
whereas for the adjoint node $r = 2$ we have the decomposition

$$B^{2,s} \cong \bigoplus_{k=0}^{s} B(k\omega_2)$$

```
sage: K = crystals.KirillovReshetikhin(['E',6,1],2,1)
sage: K.classical_decomposition()
```

Direct sum of the crystals Family (Finite dimensional highest weight crystal of type ['E → ', 6] and highest weight θ,
Finite dimensional highest weight crystal of type ['E', 6] and highest weight Lambda[2])

The promotion operator on the crystal corresponding to $\sigma$ can be calculated explicitly:

```
sage: K = crystals.KirillovReshetikhin(['E',6,1],1,1)
sage: promotion = K.promotion()
sage: u = K.module_generator(); u
[(1,)]
sage: promotion(u.lift())
[(-1, 6)]
```

The crystal $B^{1,1}$ is already of dimension 27. The elements $b$ of this crystal are labelled by tuples which specify their nonzero $\phi_i(b)$ and $\epsilon_i(b)$. For example, $[-6, 2]$ indicates that $\phi_2([-6, 2]) = \epsilon_6([-6, 2]) = 1$ and all others are equal to zero:

```
sage: K = crystals.KirillovReshetikhin(['E',6,1],1,1)
sage: K.cardinality()
27
```
Thematic Tutorials, Release 9.7

Single column KR crystals

A single column KR crystal is $B_r$ for any $r \in I_0$. 

In [LNSSS14I] and [LNSSS14II], it was shown that single column KR crystals can be constructed by projecting level 0 crystals of LS paths onto the classical weight lattice. We first verify that we do get an isomorphic crystal for $B_1$ in type $E_6^{(1)}$:

```
sage: K = crystals.KirillovReshetikhin(['E',6,1], 1,1)
sage: K2 = crystals.kirillov_reshetikhin.LSPaths(['E',6,1], 1,1)
sage: K.digraph().is_isomorphic(K2.digraph(), edge_labels=True)
True
```
Here is an example in $E_8^{(1)}$ and we calculate its classical decomposition:

```
sage: K = crystals.kirillov_reshetikhin.LSPaths(['E',8,1], 8,1)
sage: K.cardinality()
249
sage: L = [x for x in K if x.is_highest_weight([1,2,3,4,5,6,7,8])]
sage: [x.weight() for x in L]
[-2*Lambda[0] + Lambda[8], 0]
```

Applications

An important notion for finite-dimensional affine crystals is perfectness. The crucial property is that a crystal $B$ is perfect of level $\ell$ if there is a bijection between level $\ell$ dominant weights and elements in $B_{\min} = \{ b \in B \mid \text{lev}(\varphi(b)) = \ell \}$.

For a precise definition of perfect crystals see [HongKang2002]. In [FourierEtAl2010] it was proven that for the nonexceptional types $B^{r,s}$ is perfect as long as $s/c_r$ is an integer. Here $c_r = 1$ except $c_r = 2$ for $1 \leq r < n$ in type $C^{(1)}_n$ and $r = n$ in type $B^{(1)}_n$.

Here we verify this using Sage for $B^{1,1}$ of type $C^{(1)}_3$:

```
sage: K = crystals.KirillovReshetikhin(['C',3,1],1,1)
sage: Lambda = K.weight_lattice_realization().fundamental_weights(); Lambda
Finite family {0: Lambda[0], 1: Lambda[1], 2: Lambda[2], 3: Lambda[3]}
sage: [w.level() for w in Lambda]
[1, 1, 1, 1]
sage: Bmin = [b for b in K if b.Phi().level() == 1 ]; Bmin
[[[1]], [[2]], [[3]], [[-3]], [[-2]], [[-1]]]
sage: [b.Phi() for b in Bmin]
[Lambda[1], Lambda[2], Lambda[3], Lambda[2], Lambda[1], Lambda[0]]
```

As you can see, both $b = 1$ and $b = -2$ satisfy $\varphi(b) = \Lambda_1$. Hence there is no bijection between the minimal elements in $B_{\min}$ and level 1 weights. Therefore, $B^{1,1}$ of type $C^{(1)}_3$ is not perfect. However, $B^{1,2}$ of type $C^{(1)}_n$ is a perfect crystal:

```
sage: K = crystals.KirillovReshetikhin(['C',3,1],1,2)
sage: Lambda = K.weight_lattice_realization().fundamental_weights()
sage: Bmin = [b for b in K if b.Phi().level() == 1 ]
sage: [b.Phi() for b in Bmin]
[Lambda[0], Lambda[3], Lambda[2], Lambda[2], Lambda[1]]
```

Perfect crystals can be used to construct infinite-dimensional highest weight crystals and Demazure crystals using the Kyoto path model [KKKMNN1992]. We construct Example 10.6.5 in [HongKang2002]:

```
sage: K = crystals.KirillovReshetikhin(['A',1,1], 1,1)
sage: La = RootSystem(['A',1,1]).weight_lattice().fundamental_weights()
sage: B = crystals.KyotoPathModel(K, La[0])
sage: B.highest_weight_vector()
[[[2]]]
sage: K = crystals.KirillovReshetikhin(['A',2,1], 1,1)
sage: La = RootSystem(['A',2,1]).weight_lattice().fundamental_weights()
sage: B = crystals.KyotoPathModel(K, La[0])
```

(continues on next page)
Thematic Tutorials, Release 9.7

Energy function and one-dimensional configuration sum

For tensor products of Kirillov-Reshehtikhin crystals, there also exists the important notion of the energy function. It can be defined as the sum of certain local energy functions and the $R$-matrix. In Theorem 7.5 in [SchillingTingley2011] it was shown that for perfect crystals of the same level the energy $D(b)$ is the same as the affine grading (up to a normalization). The affine grading is defined as the minimal number of applications of $e_0$ to $b$ to reach a ground state path. Computationally, this algorithm is a lot more efficient than the computation involving the $R$-matrix and has been implemented in Sage:

```
sage: K = crystals.KirillovReshetikhin(['C',2,1], 2,1)
sage: La = RootSystem(['C',2,1]).weight_lattice().fundamental_weights()
sage: B = crystals.KyotoPathModel(K, La[1])
sage: B.highest_weight_vector()
[[[2], [-2]]]
```

The affine grading can be computed even for nonperfect crystals:

```
sage: K = crystals.KirillovReshetikhin(['C',4,1],1,2)
sage: K1 = crystals.KirillovReshetikhin(['C',4,1],1,1)
sage: T = crystals.TensorProduct(K,K1)
sage: hw = [b for b in T if all(b.epsilon(i)==0 for i in [1,2,3,4])]
sage: for b in hw:  
....:     print("{} {}".format(b, b.affine_grading()))
[[[], []]] 1  
[[[1, 1]], [[1]]] 2  
[[[1, 2]], [[1]]] 1  
[[[1, -1]], [[1]]] 0
```

The one-dimensional configuration sum of a crystal $B$ is the graded sum by energy of the weight of all elements $b \in B$:

$$X(B) = \sum_{b \in B} x^{weight(b)} q^{D(b)}$$

Here is an example of how you can compute the one-dimensional configuration sum in Sage:

```
sage: K = crystals.KirillovReshetikhin(['A',2,1],1,1)
sage: T = crystals.TensorProduct(K,K)
sage: T.one_dimensional_configuration_sum()
B[-2*Lambda[1] + 2*Lambda[2]] + (q+1)*B[-Lambda[1]]
```

(continues on next page)
Affine Highest Weight Crystals

Affine highest weight crystals are infinite-dimensional. Their underlying weight lattice is the extended weight lattice including the null root $\delta$. This is in contrast to finite-dimensional affine crystals such as for example the Kirillov-Reshetikhin crystals whose underlying weight lattice is the classical weight lattice, i.e., does not include $\delta$.

Hence, to work with them in Sage, we need some further tools.

Littelmann path model

The Littelmann path model for highest weight crystals is implemented in Sage. It models finite highest crystals as well as affine highest weight crystals which are infinite dimensional. The elements of the crystal are piecewise linear maps in the extended weight space over $Q$. For more information on the Littelmann path model, see [L1995].

Since the affine highest weight crystals are infinite, it is not possible to list all elements or draw the entire crystal graph. However, if the user is only interested in the crystal up to a certain distance or depth from the highest weight element, then one can work with the corresponding subcrystal. To view the corresponding upper part of the crystal, one can build the associated digraph:

```python
sage: R = RootSystem(['C', 3, 1])
sage: La = R.weight_space(extended = True).basis()
sage: LS = crystals.LSPaths(2*La[1]); LS
The crystal of LS paths of type ['C', 3, 1] and weight 2*Lambda[1]
sage: LS.weight_lattice_realization()
Extended weight space over the Rational Field of the Root system of type ['C', 3, 1]
sage: C = LS.subcrystal(max_depth=3)
sage: sorted(C, key=str)
 (-Lambda[0] + Lambda[1] + Lambda[2] - delta, Lambda[1]),
 (2*Lambda[1],),
 (Lambda[0] + Lambda[2] - Lambda[3], Lambda[1]),
 (Lambda[0] - Lambda[1] + Lambda[2], Lambda[1]),
sage: G = LS.digraph(subset = C)
sage: view(G, tightpage=True)  # optional - dot2tex graphviz, not tested (opens external..window)
```
The Littelmann path model also lends itself as a model for level zero crystals which are bi-infinite. To cut out a slice of these crystals, one can use the direction option in subcrystal:

```
sage: R = RootSystem(['A',2,1])
sage: La = R.weight_space(extended = True).basis()
sage: LS = crystals.LSPaths(La[1]-La[0]); LS
The crystal of LS paths of type ['A', 2, 1] and weight -Lambda[0] + Lambda[1]
sage: C = LS.subcrystal(max_depth=2, direction='both')
sage: G = LS.digraph(subset = C)
sage: G.set_latex_options(edge_options=lambda uv: ({}))
sage: view(G, tightpage=True) # optional - dot2tex graphviz, not tested (opens external window)
```

### Modified Nakajima monomials

Modified Nakajima monomials have also been implemented in Sage. They model highest weight crystals in all symmetric types. The elements are given in terms of commuting variables $Y_i(n)$ where $i \in I$ and $n \in \mathbb{Z}_{\geq 0}$. For more information on the modified Nakajima monomials, see [KKS2007].

We give an example in affine type and verify that up to depth 3, it agrees with the Littelmann path model. Unlike in the LS path model, the Nakajima monomial crystals are indexed by elements in the weight lattice (rather than the weight space):

```
sage: La = RootSystem(['C',3,1]).weight_space(extended = True).fundamental_weights()
sage: LS = crystals.LSPaths(2*La[1]-La[2])
sage: SL = LS.subcrystal(max_depth=3)
sage: GL = LS.digraph(subset=SL)
```

(continues on next page)
sage: La = RootSystem(['C',3,1]).weight_lattice(extended = True).fundamental_weights()

Now we do an example of a simply-laced (and hence symmetrizable) hyperbolic type $H_1^{(4)}$, which comes from the complete graph on 4 vertices:

sage: CM = CartanMatrix([[2, -1, -1,-1], [-1,2,-1,-1],[[-1,-1,2,-1],[-1,-1,-1,2]]]; CM
[[ 2 -1 -1 -1]
[-1 2 -1 -1]
[-1 -1 2 -1]
[-1 -1 -1 2]]

sage: La = RootSystem(CM).weight_lattice().fundamental_weights()

Elementary crystals

Note: Each of these crystals will work with any Cartan matrix input (with weights from the weight lattice corresponding to the Cartan matrix given).

T-crystal

Let $\lambda$ be a weight. As defined in [Kashiwara1993] (see, also, [Kashiwara1995]) the crystal $T_\lambda = \{t_\lambda\}$ is a single element crystal with the crystal structure defined by

$$\text{wt}(t_\lambda) = \lambda, \quad e_i t_\lambda = f_i t_\lambda = 0, \quad \varepsilon_i (t_\lambda) = \varphi_i (t_\lambda) = -\infty.$$ 

The crystal $T_\lambda$ shifts the weights of the vertices in a crystal $B$ by $\lambda$ when tensored with $B$, but leaves the graph structure of $B$ unchanged. That is, for all $b \in B$, we have $\text{wt}(t_\lambda \otimes b) = \text{wt}(b) + \lambda$:
Warning: Sage uses the opposite convention for the tensor product rule to Kashiwara’s definition, so care must be taken when comparing the examples here with Kashiwara’s papers.

Here is an example using a hyperbolic Cartan matrix:

```python
sage: A = CartanMatrix([[2,-4],[-4,2]])
sage: La = RootSystem(A).weight_lattice().fundamental_weights()
sage: La
Finite family {0: Lambda[0], 1: Lambda[1]}
sage: T = crystals.elementary.T(A,La[1])
sage: T
The T crystal of type [ 2 -4]
[-4  2] and weight Lambda[1]
```

C-crystal

Defined in [Kashiwara1993], the component crystal $C = \{c\}$ is the single element crystal whose crystal structure is defined by

\[
\text{wt}(c) = 0, \quad e_i c = f_i c = 0, \quad \epsilon_i(c) = \varphi_i(c) = 0.
\]

Note $C \cong B(0)$, where $B(0)$ is the highest weight crystal of highest weight $0$.

The crystal $C \otimes T_\mu$ is useful when finding subcrystals inside irreducible highest weight crystals $B(\lambda)$ where $\lambda$ is larger than $\mu$ in the lexicographic order. For example:

```python
sage: P = RootSystem("C2").weight_lattice()
sage: La = P.fundamental_weights()
sage: h = P.simple_coroots()
sage: T = crystals.elementary.T("C2", 2*La[1])
sage: C = crystals.elementary.Component(P)
```
This new crystal can be summarized into the R-crystal below.

**R-crystal**

For a fixed weight \( \lambda \), the crystal \( R_{\lambda} = \{ r_{\lambda} \} \) is a single element crystal with the crystal structure defined by

\[
\text{wt}(r_{\lambda}) = \lambda, \quad e_i r_{\lambda} = f_i r_{\lambda} = 0, \quad \epsilon_i(r_{\lambda}) = -(h_i, \lambda), \quad \varphi_i(r_{\lambda}) = 0,
\]

where \( \{ h_i \} \) are the simple coroots. See page 146 [Joseph1995], for example, for more details. (Note that in [Joseph1995], this crystal is denoted by \( S_{\lambda} \).)

Tensoring \( R_{\lambda} \) with a crystal \( B \) results in shifting the weights of the vertices in \( B \) by \( \lambda \) and may also cut a subset out of the original graph of \( B \).

**Warning:** Sage uses the opposite convention for the tensor product rule to Kashiwara’s definition, so care must be taken when comparing the examples here with some of the literature.

For example, suppose \( \mu \leq \lambda \) in lexicographic ordering on weights, and one wants to see \( B(\mu) \) as a subcrystal of \( B(\lambda) \). Then \( B(\mu) \) may be realized as the connected component of \( R_{\mu-\lambda} \otimes B(\lambda) \) containing the highest weight \( r_{\mu-\lambda} \otimes u_{\lambda} \), where \( u_{\lambda} \) is the highest weight vector in \( B(\lambda) \):

```python
sage: La = RootSystem(['B',4]).weight_lattice().fundamental_weights()
sage: Bla = crystals.NakajimaMonomials(['B',4], La[1]+La[2])
sage: Bmu = crystals.NakajimaMonomials(['B',4], La[1])
sage: R = crystals.elementary.R(['B',4], -La[2])
sage: T = crystals.TensorProduct(R,Bla)
sage: mg = mg = T(R[0], Bla.module_generators[0])
sage: S = T.subcrystal(generators=[mg])
sage: G = T.digraph(subset=S)
sage: Bmu.digraph().is_isomorphic(G, edge_labels=True)
True
sage: view(G, tightpage=True)  # optional - dot2tex graphviz, not tested (opens external window)
```
**i-th elementary crystal**

For \( i \) an element of the index set of type \( X \), the crystal \( B_i \) of type \( X \) is the set

\[
B_i = \{ b_i(m) : m \in \mathbb{Z} \},
\]

where the crystal structure is given by

\[
\varphi_j(b_i(m)) = \begin{cases} 
  m & \text{if } j = i, \\
  -\infty & \text{if } j \neq i,
\end{cases} \quad \varepsilon_j(b_i(m)) = \begin{cases} 
  -m & \text{if } j = i, \\
  -\infty & \text{if } j \neq i,
\end{cases}
\]

\[
f_j(b_i(m)) = \begin{cases} 
  b_i(m-1) & \text{if } j = i, \\
  0 & \text{if } j \neq i,
\end{cases} \quad e_j(b_i(m)) = \begin{cases} 
  b_i(m+1) & \text{if } j = i, \\
  0 & \text{if } j \neq i.
\end{cases}
\]

See [Kashiwara1993] or [Kashiwara1995] for more information. Here is an example:

```python
sage: B = crystals.elementary.Elementary("A2", 1)
sage: S = B.subcrystal(max_depth=4, generators=[B(0)])
sage: sorted(s for s in S)
[-4, -3, -2, -1, 0, 1, 2, 3, 4]
sage: G = B.digraph(subset=S)
sage: view(G, tightpage=True)  # optional - dot2tex graphviz, not tested (opens external window)
```
Warning: To reiterate, Sage uses the opposite convention for the tensor product rule to Kashiwara’s definition. In particular, using Sage’s convention, one has $T_{\lambda} \otimes B_i \cong B_i \otimes T_{s_i \lambda}$, where $s_i$ is the $i$-th simple reflection.

Infinity Crystals

Infinity crystals, denoted $\mathcal{B}(\infty)$ and depend only on the Cartan type, are the crystal analogue of Verma modules with highest weight $0$ associated to a symmetrizable Kac-Moody algebra (equivalently, the lower half of the corresponding quantum group). As such, they are infinite-dimensional and any irreducible highest weight crystal may be obtained from an infinity crystal via some cutting procedure. On the other hand, the crystal $\mathcal{B}(\infty)$ is the direct limit of all irreducible highest weight crystals $\mathcal{B}(\lambda)$, so there are natural embeddings of each $\mathcal{B}(\lambda)$ in $\mathcal{B}(\infty)$. Below, we outline the various implementations of the crystal $\mathcal{B}(\infty)$ in Sage and give examples of how $\mathcal{B}(\lambda)$ interacts with $\mathcal{B}(\infty)$.

All infinity crystals that are currently implemented in Sage can be accessed by typing `crystals.infinity.<tab>`.

Marginally large tableaux

Marginally large tableaux were introduced by J. Hong and H. Lee as a realization of the crystal $\mathcal{B}(\infty)$ in types $A_n$, $B_n$, $C_n$, $D_{n+1}$, and $G_2$. The marginally large condition guarantees that all tableau have exactly $n$ rows and that the number of $i$-boxes in the $i$-th row from the top (in the English convention) is exactly one more than the total number of boxes in the $(i + 1)$-st row. Other specific conditions on the tableaux vary by type. See [HongLee2008] for more information.

Here is an example in type $A_2$:

```python
sage: B = crystals.infinity.Tableaux(['A',2])
sage: b = B.highest_weight_vector()
```
Since the crystal is infinite, we must specify the subcrystal we would like to view:

```
sage: B = crystals.infinity.Tableaux(['A',2])
sage: S = B.subcrystal(max_depth=4)
sage: G = B.digraph(subset=S)
sage: view(G, tightpage=True)  # optional - dot2tex graphviz, not tested (opens external window)
```

Using elementary crystals, we can cut irreducible highest weight crystals from $B(\infty)$ as the connected component of $C \otimes T_\lambda \otimes B(\infty)$ containing $c \otimes t_\lambda \otimes u_\infty$, where $u_\infty$ is the highest weight vector in $B(\infty)$. In this example, we cut out $B(\rho)$ from $B(\infty)$ in type $A_2$:

```
sage: B = crystals.infinity.Tableaux(['A',2])
sage: b = B.highest_weight_vector()
sage: T = crystals.elementary.T(['A',2], B.Lambda()[1] + B.Lambda()[2])
sage: t = T[0]
sage: C = crystals.elementary.Component(['A',2])
sage: c = C[0]
sage: TP = crystals.TensorProduct(C,T,B)
sage: t0 = TP(c,t,b)
sage: STP = TP.subcrystal(generators=[t0])
sage: GTP = TP.digraph(subset=STP)
sage: view(GTP, tightpage=True)  # optional - dot2tex graphviz, not tested (opens external window)
```
Note that the above code can be simplified using the R-crystal:

```python
sage: B = crystals.infinity.Tableaux(['A',2])
sage: b = B.highest_weight_vector()
sage: R = crystals.elementary.R(['A',2], B.Lambda()[1] + B.Lambda()[2])
sage: r = R[0]
sage: TP2 = crystals.TensorProduct(R,B)
sage: t2 = TP2(r,b)
sage: STP2 = TP2.subcrystal(generators=[t2])
sage: GTP2 = TP2.digraph(subset=STP2)
sage: view(GTP2, tightpage=True) # optional - dot2tex graphviz, not tested (opens external window)
```

On the other hand, we can embed the irreducible highest weight crystal $B(\rho)$ into $B(\infty)$:

```python
sage: Brho = crystals.Tableaux(['A',2],shape=[2,1])
sage: brho = Brho.highest_weight_vector()
sage: B = crystals.infinity.Tableaux(['A',2])
sage: binf = B.highest_weight_vector()
sage: wt = brho.weight()
```

(continues on next page)
sage: T = crystals.elementary.T(['A', 2], wt)
sage: TlambdaBinf = crystals.TensorProduct(T, B)
sage: hw = TlambdaBinf(T[0], binf)
sage: Psi = Brho.crystal_morphism({brho : hw})
sage: BG = B.digraph(subset=[Psi(x) for x in Brho])
sage: view(BG, tightpage=True) # optional - dot2tex graphviz, not tested (opens external window)

Note that in the last example, we had to inject $B(\rho)$ into the tensor product $T_{\lambda} \otimes B(\infty)$, since otherwise, the map $\Psi$ would not be a crystal morphism (as $b.weight() \neq brho.weight()$).

### Generalized Young walls

Generalized Young walls were introduced by J.-A. Kim and D.-U. Shin as a model for $B(\infty)$ and each $B(\lambda)$ solely in affine type $A_n^{(1)}$. See [KimShin2010] for more information on the construction of generalized Young walls.

Since this model is only valid for one Cartan type, the input to initialize the crystal is simply the rank of the underlying type:

sage: Y = crystals.infinity.GeneralizedYoungWalls(2)
sage: y = Y.highest_weight_vector()
sage: y.f_string([0, 1, 2, 2, 2, 1, 0, 0, 1, 2]).pp()

```
2|   |
  |   |
1|2|   |
  0|1|
2|0|1|2|0|
```

In the `weight` method for this model, we can choose whether to view weights in the extended weight lattice (by default) or in the root lattice:

sage: Y = crystals.infinity.GeneralizedYoungWalls(2)
sage: y = Y.highest_weight_vector()
sage: y.f_string([0, 1, 2, 2, 2, 1, 0, 0, 1, 2]).weight()
\[
\text{Lambda}[0] + \text{Lambda}[1] - 2*\text{Lambda}[2] - 3*\delta
\]

\[
\text{sage:} \quad y.f_string([0,1,2,2,1,0,1,2]).weight(\text{root lattice=True})
\]

\[
\]

As before, we need to indicate a specific subcrystal when attempting to view the crystal graph:

\[
\text{sage:} \quad Y = \text{crystals.infinity.GeneralizedYoungWalls(2)}
\]

\[
\text{sage:} \quad SY = Y.\text{subcrystal}(\text{max_depth=3})
\]

\[
\text{sage:} \quad GY = Y.\text{digraph}(\text{subset=SY})
\]

\[
\text{sage:} \quad \text{view(GY, tightpage=True) \# optional - dot2tex graphviz, not tested (opens external window)}
\]

One can also make irreducible highest weight crystals using generalized Young walls:

\[
\text{sage:} \quad L = \text{RootSystem(['A',2,1]).weight lattice(extended=True).fundamental weights()}
\]

\[
\text{sage:} \quad YL = \text{crystals.GeneralizedYoungWalls(2,L[0])}
\]

\[
\text{sage:} \quad SYL = YL.\text{subcrystal}(\text{max_depth=3})
\]

\[
\text{sage:} \quad GYL = YL.\text{digraph}(\text{subset=SYL})
\]

\[
\text{sage:} \quad \text{view(GYL, tightpage=True) \# optional - dot2tex graphviz, not tested (opens external window)}
\]

In the highest weight crystals, however, weights are always elements of the extended affine weight lattice:

\[
\text{sage:} \quad L = \text{RootSystem(['A',2,1]).weight lattice(extended=True).fundamental weights()}
\]

\[
\text{sage:} \quad YL = \text{crystals.StandardYoungWalls(2,L[0])}
\]

\[
\text{sage:} \quad YL.\text{highest weight vector()}.f_string([0,1,2,0]).weight()
\]

\[
-\text{Lambda}[0] + \text{Lambda}[1] + \text{Lambda}[2] - 2*\delta
\]
Modified Nakajima monomials

Let $Y_{i,k}$, for $i \in I$ and $k \in \mathbb{Z}$, be a commuting set of variables, and let $1$ be a new variable which commutes with each $Y_{i,k}$. (Here, $I$ represents the index set of a Cartan datum.) One may endow the structure of a crystal on the set $\hat{\mathcal{M}}$ of monomials of the form

$$M = \prod_{(i,k) \in I \times \mathbb{Z}_{\geq 0}} Y_{i,k}^{y_{i,k}(k)}1.$$  

Elements of $\hat{\mathcal{M}}$ are called modified Nakajima monomials. We will omit the $1$ from the end of a monomial if there exists at least one $y_{i,k}(k) \neq 0$. The crystal structure on this set is defined by

\begin{align*}
\text{wt}(M) &= \sum_{i \in I} \left( \sum_{k \geq 0} y_{i,k}(k) \right) \Lambda_i, \\
\varphi_i(M) &= \max \left\{ \sum_{0 \leq j \leq k} y_{i,j}(j) : k \geq 0 \right\}, \\
\varepsilon_i(M) &= \varphi_i(M) - \langle h_i, \text{wt}(M) \rangle, \\
k_f &= k_f(M) = \min \left\{ k \geq 0 : \varphi_i(M) = \sum_{0 \leq j \leq k} y_{i,j}(j) \right\}, \\
k_e &= k_e(M) = \max \left\{ k \geq 0 : \varphi_i(M) = \sum_{0 \leq j \leq k} y_{i,j}(j) \right\},
\end{align*}

where $\{h_i : i \in I\}$ and $\{\Lambda_i : i \in I\}$ are the simple coroots and fundamental weights, respectively. With a chosen set of integers $C = (c_{ij})_{i \neq j}$ such that $c_{ij} + c_{ji} = 1$, one defines

$$A_{i,k} = Y_{i,k}Y_{i,k+1} \prod_{j \neq i} Y_{j,k+c_{ij}},$$

where $(a_{ij})$ is a Cartan matrix. Then

\begin{align*}
\varepsilon_i M &= \begin{cases} 0 & \text{if } \varepsilon_i(M) = 0, \\ A_{i,k} M & \text{if } \varepsilon_i(M) > 0, \end{cases} \\
f_i M &= A_{i,k}^{-1} M.
\end{align*}

Note: Monomial crystals depend on the choice of positive integers $C = (c_{ij})_{i \neq j}$ satisfying the condition $c_{ij} + c_{ji} = 1$. This choice has been made in Sage such that $c_{ij} = 1$ if $i < j$ and $c_{ij} = 0$ if $i > j$, but other choices may be used if deliberately stated at the initialization of the crystal:

```
sage: c = Matrix([[0,0,1],[1,0,0],[0,1,0]])
sage: La = RootSystem(['C',3]).weight_lattice().fundamental_weights()
sage: M = crystals.NakajimaMonomials(2*La[1], c=c)
sage: M.c()
[0 0 1]
[1 0 0]
[0 1 0]
```

It is shown in [KKS2007] that the connected component of $\hat{\mathcal{M}}$ containing the element $1$, which we denote by $\mathcal{M}(\infty)$, is crystal isomorphic to the crystal $B(\infty)$.
```
sage: Minf = crystals.infinity.NakajimaMonomials(["C",3,1])
sage: minf = Minf.highest_weight_vector()
sage: m = minf.f_string([0,1,2,3,2,1,0]); m
  Y(0,0)^-1 Y(0,4)^-1 Y(1,0) Y(1,3)
sage: m.weight()
sage: m.weight_in_root_lattice()
```

We can also model $B(\infty)$ using the variables $A_{i,k}$ instead:

```
sage: Minf = crystals.infinity.NakajimaMonomials(["C",3,1])
sage: minf = Minf.highest_weight_vector()
sage: Minf.set_variables('A')
sage: m = minf.f_string([0,1,2,3,2,1,0]); m
  A(0,0)^-1 A(0,3)^-1 A(1,0)^-1 A(1,2)^-1 A(2,0)^-1 A(2,1)^-1 A(3,0)^-1
sage: m.weight()
sage: m.weight_in_root_lattice()
sage: Minf.set_variables('Y')
```

Building the crystal graph output for these monomial crystals is the same as the constructions above:

```
sage: Minf = crystals.infinity.NakajimaMonomials(["C",3,1])
sage: Sinf = Minf.subcrystal(max_depth=2)
sage: Ginf = Minf.digraph(subset=Sinf)
sage: view(Ginf, tightpage=True) # optional - dot2tex graphviz, not tested (opens external window)
```

Note that this model will also work for any symmetrizable Cartan matrix:

```
sage: A = CartanMatrix([[2,-4],[-5,2]])
sage: Linf = crystals.infinity.NakajimaMonomials(A); Linf
  Infinity Crystal of modified Nakajima monomials of type [ 2 -4]
  [-5 2]
sage: Linf.highest_weight_vector().f_string([0,1,1,1,0,0,1,1,0])
  Y(0,0)^-1 Y(0,1)^9 Y(0,2)^5 Y(0,3)^-1 Y(1,0)^2 Y(1,1)^5 Y(1,2)^3
```
Rigged configurations

Rigged configurations are sequences of partitions, one partition for each node in the underlying Dynkin diagram, such that each part of each partition has a label (or rigging). A crystal structure was defined on these objects in [Schilling2006], then later extended to work as a model for $B(\infty)$. See [SalisburyScrimshaw2015] for more information:

```
sage: RiggedConfigurations.options(display="horizontal")
sage: RC = crystals.infinity.RiggedConfigurations(['C',3,1])
sage: nu = RC.highest_weight_vector().f_string([0,1,2,3,2,1,0]); nu
-2[ ]-1 2[ ]1 0[ ]0 0[ ]0
-2[ ]-1 2[ ]1 0[ ]0
sage: nu.weight()
sage: RiggedConfigurations.options._reset()
```

We can check this crystal is isomorphic to the crystal above using Nakajima monomials:

```
sage: Minf = crystals.infinity.NakajimaMonomials(['C',3,1])
sage: Sinf = Minf.subcrystal(max_depth=2)
sage: Ginf = Minf.digraph(subset=Sinf)
sage: RC = crystals.infinity.RiggedConfigurations(['C',3,1])
sage: RCS = RC.subcrystal(max_depth=2)
sage: RCG = RC.digraph(subset=RCS)
sage: RCG.is_isomorphic(Ginf, edge_labels=True)
True
```

This model works in Sage for all finite and affine types, as well as any simply laced Cartan matrix.

Iwahori Hecke Algebras

The Iwahori Hecke algebra is defined in [Iwahori1964]. In that original paper, the algebra occurs as the convolution ring of functions on a $p$-adic group that are compactly supported and invariant both left and right by the Iwahori subgroup. However Iwahori determined its structure in terms of generators and relations, and it turns out to be a deformation of the group algebra of the affine Weyl group.

Once the presentation is found, the Iwahori Hecke algebra can be defined for any Coxeter group. It depends on a parameter $q$ which in Iwahori’s paper is the cardinality of the residue field. But it could just as easily be an indeterminate.

Then the Iwahori Hecke algebra has the following description. Let $W$ be a Coxeter group, with generators (simple reflections) $s_1, \ldots, s_n$. They satisfy the relations $s_i^2 = 1$ and the braid relations

$$s_is_js_i\cdots = s_js_is_i\cdots$$

where the number of terms on each side is the order of $s_is_j$.

The Iwahori Hecke algebra has a basis $T_1, \ldots, T_n$ subject to relations that resemble those of the $s_i$. They satisfy the braid relations and the quadratic relation

$$(T_i - q)(T_i + 1) = 0.$$  

This can be modified by letting $q_1$ and $q_2$ be two indeterminates and letting

$$(T_i - q_1)(T_i - q_2) = 0.$$  

In this generality, Iwahori Hecke algebras have significance far beyond their origin in the representation theory of $p$-adic groups. For example, they appear in the geometry of Schubert varieties, where they are used in the definition of the Kazhdan-Lusztig polynomials. They appear in connection with quantum groups, and in Jones’s original paper on the Jones polynomial.

Here is how to create an Iwahori Hecke algebra (in the $T$ basis):

```
sage: R.<q> = PolynomialRing(ZZ)
sage: H = IwahoriHeckeAlgebra("B3",q)
sage: T = H.T(); T
Iwahori-Hecke algebra of type B3 in q,-1 over Univariate Polynomial Ring
   in q over Integer Ring in the T-basis
sage: T1,T2,T3 = T.algebra_generators()
sage: T1*T1
(q-1)*T[1] + q
```

If the Cartan type is affine, the generators will be numbered starting with $T_0$ instead of $T_1$.

You may convert a Weyl group element into the Iwahori Hecke algebra:

```
sage: W = WeylGroup("G2",prefix="s")
sage: [s1,s2] = W.simple_reflections()
sage: P.<q> = LaurentPolynomialRing(QQ)
sage: H = IwahoriHeckeAlgebra("G2",q)
sage: T = H.T()
sage: T(s1*s2)
T[1,2]
```

## Kazhdan-Lusztig Polynomials

Sage can compute ordinary Kazhdan-Lusztig polynomials for Weyl groups or affine Weyl groups (and potentially other Coxeter groups).

You must create a Weyl group $W$ and a ring containing an indeterminate $q$. The ring may be a univariate polynomial ring or a univariate Laurent polynomial ring. Then you may calculate Kazhdan-Lusztig polynomials as follows:

```
sage: W = WeylGroup("A3", prefix="s")
sage: [s1,s2,s3] = W.simple_reflections()
sage: P.<q> = LaurentPolynomialRing(QQ)
sage: KL = KazhdanLusztigPolynomial(W,q)
sage: KL.R(s2, s2*s1*s3*s2)
-1 + 3*q - 3*q^2 + q^3
sage: KL.P(s2, s2*s1*s3*s2)
1 + q
```

Thus we have the Kazhdan-Lusztig $R$ and $P$ polynomials.

Known algorithms for computing Kazhdan-Lusztig polynomials are highly recursive, and caching of intermediate results is necessary for the programs not to be prohibitively slow. Therefore intermediate results are cached. This has the effect that as you run the program for any given KazhdanLusztigPolynomial class, the calculations will be slow at first but progressively faster as more polynomials are computed.

You may see the results of the intermediate calculations by creating the class with the option `trace="true"`.

Since the parent of $q$ must be a univariate ring, if you want to work with other indeterminates, first create a univariate polynomial or Laurent polynomial ring, and the Kazhdan-Lusztig class. Then create a ring containing $q$ and the other variables:
```python
sage: W = WeylGroup("B3", prefix="s")
sage: [s1,s2,s3] = W.simple_reflections()
sage: P.<q> = PolynomialRing(QQ)
sage: KL = KazhdanLusztigPolynomial(W,q)
sage: P1.<x,y> = PolynomialRing(P)
sage: x*KL.P(s1*s3,s1*s3*s2*s1*s3)
(q + 1)*x
```

### Bibliography

Preparation of this document was supported in part by NSF grants DMS-0652817, DMS-1001079, OCI-1147463, DMS-0652641, DMS-0652652, DMS-1001256, OCI-1147247 and DMS-1601026.

### 12.1.5 Tutorial: Vector Calculus in Euclidean Spaces

**Author: Eric Gourgoulhon**

This document contains various tutorials introducing vector calculus with SageMath. The first one regards vector calculus in the 3-dimensional Euclidean space \( \mathbb{E}^3 \) in Cartesian coordinates, focusing on the evaluation of the standard vector operators. The second tutorial deals with the same topic but based on curvilinear (spherical and cylindrical) coordinates. The third tutorial is devoted to changes between the various coordinate systems. The fourth tutorial presents some advanced aspects, namely the treatment of \( \mathbb{E}^3 \) as a Riemannian manifold. Finally, the last tutorial is devoted to 2-dimensional vector calculus, using both Cartesian and polar coordinates in the Euclidean plane \( \mathbb{E}^2 \); it combines various features of the other tutorials.

#### How to compute a gradient, a divergence or a curl

This tutorial introduces some vector calculus capabilities of SageMath within the 3-dimensional Euclidean space. The corresponding tools have been developed via the SageManifolds project.

The tutorial is also available as a Jupyter notebook, either passive (nbviewer) or interactive (binder).

#### First stage: introduce the Euclidean 3-space

Before evaluating some vector-field operators, one needs to define the arena in which vector fields live, namely the 3-dimensional Euclidean space \( \mathbb{E}^3 \). In SageMath, we declare it, along with the standard Cartesian coordinates \((x, y, z)\), via EuclideanSpace:

```python
sage: E.<x,y,z> = EuclideanSpace()
sage: E
Euclidean space \mathbb{E}^3
```

Thanks to the notation \( <x,y,z> \) in the above declaration, the coordinates \((x, y, z)\) are immediately available as three symbolic variables \(x, y\) and \(z\) (there is no need to declare them via `var()`; i.e. to type \(x, y, z = var('x y z')\)):
Besides, $\mathbb{E}^3$ is endowed with the orthonormal vector frame $(e_x, e_y, e_z)$ associated with Cartesian coordinates:

```
sage: E.frames()
[Coordinate frame (E^3, (e_x,e_y,e_z))]
```

At this stage, this is the default vector frame on $\mathbb{E}^3$ (being the only vector frame introduced so far):  

```
sage: E.default_frame()
Coordinate frame (E^3, (e_x,e_y,e_z))
```

---

**Defining a vector field**

We define a vector field on $\mathbb{E}^3$ from its components in the vector frame $(e_x, e_y, e_z)$:

```
sage: v = E.vector_field(-y, x, sin(x*y*z), name='v')
sage: v.display()
v = -y e_x + x e_y + sin(x*y*z) e_z
```

We can access to the components of $v$ via the square bracket operator:

```
sage: v[1]
-y
sage: v[:]
[-y, x, sin(x*y*z)]
```

A vector field can evaluated at any point of $\mathbb{E}^3$:

```
sage: p = E((3,-2,1), name='p')
sage: p
Point p on the Euclidean space E^3
sage: p.coordinates()
(3, -2, 1)
sage: vp = v.at(p)
sage: vp
Vector v at Point p on the Euclidean space E^3
sage: vp.display()
v = 2 e_x + 3 e_y - sin(6) e_z
```

Vector fields can be plotted:

```
sage: v.plot(max_range=1.5, scale=0.5)
Graphics3d Object
```

For customizing the plot, see the list of options in the documentation of `plot()`. For instance, to get a view of the orthogonal projection of $v$ in the plane $y = 1$, do:

---

**12.1. Thematic tutorial document tree**

227
We may define a vector field \( u \) with generic components \((u_x, u_y, u_z)\):

\[
\text{sage: } u = E\text{.vector_field(function('u_x')(x,y,z),}
\quad \text{function('u_y')(x,y,z),}
\quad \text{function('u_z')(x,y,z),}
\quad \text{name='u')}\n\]

\[
\text{sage: } u\text{.display()}
\]

\[
u = u_x(x, y, z)\ e_x + u_y(x, y, z)\ e_y + u_z(x, y, z)\ e_z
\]

\[
\text{sage: } u[:]
\]

\[
[u_x(x, y, z), u_y(x, y, z), u_z(x, y, z)]
\]

Its value at the point \( p \) is then:

\[
\text{sage: } u_p = u\text{.at(p)}
\]

\[
\text{sage: } u_p\text{.display()}
\]

\[
u = u_x(3, -2, 1)\ e_x + u_y(3, -2, 1)\ e_y + u_z(3, -2, 1)\ e_z
\]
How to compute various vector products

Dot product

The dot (or scalar) product $\mathbf{u} \cdot \mathbf{v}$ of the vector fields $\mathbf{u}$ and $\mathbf{v}$ is obtained by the method `dot_product()`, which admits `dot()` as a shortcut alias:

```
True
```

$s = \mathbf{u} \cdot \mathbf{v}$ is a scalar field, i.e. a map $\mathbb{E}^3 \to \mathbb{R}$:

```
sage: s = u.dot(v)
sage: s
Scalar field u.v on the Euclidean space $\mathbb{E}^3$
sage: s.display()
u.v: $\mathbb{E}^3 \to \mathbb{R}$
    (x, y, z) ↦ -y*u_x(x, y, z) + x*u_y(x, y, z) + sin(x*y*z)*u_z(x, y, z)
```

It maps points of $\mathbb{E}^3$ to real numbers:

```
sage: s(p)
-sin(6)*u_z(3, -2, 1) + 2*u_x(3, -2, 1) + 3*u_y(3, -2, 1)
```

Its coordinate expression is:

```
sage: s.expr()
-y*u_x(x, y, z) + x*u_y(x, y, z) + sin(x*y*z)*u_z(x, y, z)
```

Norm

The norm $\|\mathbf{u}\|$ of the vector field $\mathbf{u}$ is defined in terms of the dot product by $\|\mathbf{u}\| = \sqrt{\mathbf{u} \cdot \mathbf{u}}$:

```
sage: norm(u) == sqrt(u.dot(u))
True
```

It is a scalar field on $\mathbb{E}^3$:

```
sage: s = norm(u)
sage: s
Scalar field $|\mathbf{u}|$ on the Euclidean space $\mathbb{E}^3$
sage: s.display()
|\mathbf{u}|: $\mathbb{E}^3 \to \mathbb{R}$
    (x, y, z) ↦ sqrt(u_x(x, y, z)^2 + u_y(x, y, z)^2 + u_z(x, y, z)^2)
sage: s.expr()
sqrt(u_x(x, y, z)^2 + u_y(x, y, z)^2 + u_z(x, y, z)^2)
```

For $\mathbf{v}$, we have:

```
sage: norm(v).expr()
sqrt(x^2 + y^2 + sin(x*y*z)^2)
```
Cross product

The cross product \( u \times v \) is obtained by the method `cross_product()`, which admits `cross()` as a shortcut alias:

```
sage: s = u.cross(v)
sage: s
Vector field u x v on the Euclidean space E^3
sage: s.display()
u x v = (\sin(x^y^z) u_y(x, y, z) - x u_z(x, y, z)) e_x
  + (-\sin(x^y^z) u_x(x, y, z) - y u_z(x, y, z)) e_y
  + (x u_x(x, y, z) + y u_y(x, y, z)) e_z
```

We can check the standard formulas expressing the cross product in terms of the components:

```
True
```

Scalar triple product

Let us introduce a third vector field, \( w \) say. As an example, we do not pass the components as arguments of `vector_field()`, as we did for \( u \) and \( v \); instead, we set them in a second stage, via the square bracket operator, any unset component being assumed to be zero:

```
sage: w = E.vector_field(name='w')
sage: w[1] = x*z
sage: w[2] = y*z
sage: w.display()
w = x*z e_x + y*z e_y
```

The scalar triple product of the vector fields \( u, v \) and \( w \) is obtained as follows:

```
sage: triple_product = E.scalar_triple_product()
sage: s = triple_product(u, v, w)
sage: s
Scalar field epsilon(u,v,w) on the Euclidean space E^3
sage: s.expr()
-(y u_x(x, y, z) - x u_y(x, y, z)) z sin(x^y^z) - (x^2 u_z(x, y, z)
  + y^2 u_x(x, y, z)) z
```

Let us check that the scalar triple product of \( u, v \) and \( w \) is \( u \cdot (v \times w) \):

```
sage: s == u.dot(v.cross(w))
True
```
How to evaluate the standard differential operators

The standard operators grad, div, curl, etc. involved in vector calculus are accessible as methods on scalar fields and vector fields (e.g. v.div()). However, to allow for standard mathematical notations (e.g. div(v)), let us import the functions grad(), div(), curl() and laplacian():

```
sage: from sage.manifolds.operators import *
```

Gradient

We first introduce a scalar field, via its expression in terms of Cartesian coordinates; in this example, we consider some unspecified function of \((x, y, z)\):

```
sage: F = E.scalar_field(function('f')(x,y,z), name='F')
sage: F.display()
F: E^3 \rightarrow \mathbb{R}
  \( (x, y, z) \mapsto f(x, y, z) \)
```

The value of \(F\) at a point:

```
sage: F(p)
f(3, -2, 1)
```

The gradient of \(F\):

```
sage: grad(F)
Vector field grad(F) on the Euclidean space E^3
sage: grad(F).display()
grad(F) = \frac{\partial f}{\partial x} e_x + \frac{\partial f}{\partial y} e_y + \frac{\partial f}{\partial z} e_z
sage: norm(grad(F)).display()
|grad(F)|: E^3 \rightarrow \mathbb{R}
  \( (x, y, z) \mapsto \sqrt{(\frac{\partial f}{\partial x})^2 + (\frac{\partial f}{\partial y})^2 + (\frac{\partial f}{\partial z})^2} \)
```

Divergence

The divergence of the vector field \(u\):

```
sage: s = div(u)
sage: s.display()
div(u): E^3 \rightarrow \mathbb{R}
  \( (x, y, z) \mapsto \frac{\partial u_x}{\partial x} + \frac{\partial u_y}{\partial y} + \frac{\partial u_z}{\partial z} \)
```

For \(v\) and \(w\), we have:

```
sage: div(v).expr()
x*y*cos(x*y*z)
sage: div(w).expr()
2*z
```

An identity valid for any scalar field \(F\) and any vector field \(u\):
sage: div(F*u) == F*div(u) + u.dot(grad(F))
True

Curl

The curl of the vector field \( \mathbf{u} \):

sage: s = curl(u)
sage: s
Vector field curl(u) on the Euclidean space E^3
sage: s.display()
curl(u) = (-d(u_y)/dz + d(u_z)/dy) \mathbf{e}_x + (d(u_x)/dz - d(u_z)/dx) \mathbf{e}_y + (-d(u_x)/dy + d(u_y)/dx) \mathbf{e}_z

To use the notation rot instead of curl, simply do:

sage: rot = curl

An alternative is:

sage: from sage.manifolds.operators import curl as rot

We have then:

sage: rot(u).display()
curl(u) = (-d(u_y)/dz + d(u_z)/dy) \mathbf{e}_x + (d(u_x)/dz - d(u_z)/dx) \mathbf{e}_y + (-d(u_x)/dy + d(u_y)/dx) \mathbf{e}_z
sage: rot(u) == curl(u)
True

For \( \mathbf{v} \) and \( \mathbf{w} \), we have:

sage: curl(v).display()
curl(v) = x*z*cos(x*y*z) \mathbf{e}_x - y*z*cos(x*y*z) \mathbf{e}_y + 2 \mathbf{e}_z
sage: curl(w).display()
curl(w) = -y \mathbf{e}_x + x \mathbf{e}_y

The curl of a gradient is always zero:

sage: curl(grad(F)).display()
curl(grad(F)) = 0

The divergence of a curl is always zero:

sage: div(curl(u)).display()
div(curl(u)): E^3 \to \mathbb{R}
\( (x, y, z) \mapsto 0 \)

An identity valid for any scalar field \( F \) and any vector field \( \mathbf{u} \) is

\[
\text{curl}(F\mathbf{u}) = \text{grad} F \times \mathbf{u} + F \text{curl} \mathbf{u},
\]

as we can check:
Laplacian

The Laplacian $\Delta F$ of a scalar field $F$ is another scalar field:

```
sage: s = laplacian(F)
sage: s.display()
\Delta F: E^3 \rightarrow \mathbb{R}
\quad (x, y, z) \mapsto \frac{d^2(f)}{dx^2} + \frac{d^2(f)}{dy^2} + \frac{d^2(f)}{dz^2}
```

The following identity holds:

$$\Delta F = \text{div} (\text{grad } F),$$

as we can check:

```
sage: laplacian(F) == div(grad(F))
True
```

The Laplacian $\Delta u$ of a vector field $u$ is another vector field:

```
sage: Du = laplacian(u)
sage: Du
Vector field Delta(u) on the Euclidean space $E^3$
```

whose components are:

```
sage: Du.display()
\Delta u = \frac{d^2(u_x)}{dx^2} + \frac{d^2(u_x)}{dy^2} + \frac{d^2(u_x)}{dz^2} \, e_x
+ \frac{d^2(u_y)}{dx^2} + \frac{d^2(u_y)}{dy^2} + \frac{d^2(u_y)}{dz^2} \, e_y
+ \frac{d^2(u_z)}{dx^2} + \frac{d^2(u_z)}{dy^2} + \frac{d^2(u_z)}{dz^2} \, e_z
```

In the Cartesian frame, the components of $\Delta u$ are nothing but the (scalar) Laplacians of the components of $u$, as we can check:

```
sage: e = E.cartesian_frame()
sage: Du == sum(laplacian(u[[i]])*e[i] for i in E.irange())
True
```

In the above formula, $u[[i]]$ return the $i$-th component of $u$ as a scalar field, while $u[i]$ would have returned the coordinate expression of this scalar field; besides, $e$ is the Cartesian frame:

```
sage: e[:]
(Vector field e_x on the Euclidean space $E^3$,
Vector field e_y on the Euclidean space $E^3$,
Vector field e_z on the Euclidean space $E^3$)
```

For the vector fields $v$ and $w$, we have:
sage: laplacian(v).display()
\Delta(v) = -(x^2*y^2 + (x^2 + y^2)z^2)*\sin(x*y*z) \, e_z

sage: laplacian(w).display()
\Delta(w) = 0

We have:

sage: curl(curl(u)).display()
curl(curl(u)) = (d^2(u_x)/dy^2 - d^2(u_x)/dz^2 + d^2(u_y)/dxdy
+ d^2(u_z)/dxdz) \, e_x + (d^2(u_x)/dxdy - d^2(u_y)/dx^2 - d^2(u_y)/dz^2
+ d^2(u_z)/dydz) \, e_y + (d^2(u_x)/dxdz + d^2(u_y)/dydz - d^2(u_z)/dx^2
- d^2(u_z)/dy^2) \, e_z

sage: grad(div(u)).display()
grad(div(u)) = (d^2(u_x)/dx^2 + d^2(u_y)/dxdy + d^2(u_z)/dxdz) \, e_x
+ (d^2(u_x)/dxdy + d^2(u_y)/dy^2 + d^2(u_z)/dydz) \, e_y
+ (d^2(u_x)/dxdz + d^2(u_y)/dydz + d^2(u_z)/dz^2) \, e_z

A famous identity is

\text{curl (curl } u) = \text{grad (div } u) - \Delta u.\\

Let us check it:

sage: curl(curl(u)) == grad(div(u)) - laplacian(u)
True

How to customize various symbols

Customizing the symbols of the orthonormal frame vectors

By default, the vectors of the orthonormal frame associated with Cartesian coordinates are denoted \((e_x, e_y, e_z)\):

sage: frame = E.cartesian_frame()
sage: frame
Coordinate frame (E^3, (e_x,e_y,e_z))

But this can be changed, thanks to the method \text{set_name()}:

sage: frame.set_name('a', indices=('x', 'y', 'z'))
sage: frame
Coordinate frame (E^3, (a_x,a_y,a_z))

sage: v.display()
v = -y \, a_x + x \, a_y + \sin(x*y*z) \, a_z

sage: frame.set_name(('hx', 'hy', 'hz'), latex_symbol=(r'\mathbf{\hat{x}}', r'\mathbf{\hat{y}}', r'\mathbf{\hat{z}}'))

sage: frame
Coordinate frame (E^3, (hx,hy,hz))

sage: v.display()
v = -y \, hx + x \, hy + \sin(x*y*z) \, hz
Customizing the coordinate symbols

The coordinates symbols are defined within the angle brackets <...> at the construction of the Euclidean space. Above we did:

```sage
E.<x,y,z> = EuclideanSpace()
```

which resulted in the coordinate symbols \((x, y, z)\) and in the corresponding Python variables \(x, y\) and \(z\) (SageMath symbolic expressions). To use other symbols, for instance \((X, Y, Z)\), it suffices to create \(E\) as:

```sage
E.<X,Y,Z> = EuclideanSpace()
```

We have then:

```sage
E.atlas()
[Chart (E^3, (X, Y, Z))]
E.cartesian_frame()
Coordinate frame (E^3, (e_X,e_Y,e_Z))
v = E.vector_field(-Y, X, sin(X*Y*Z), name='v')
v.display()
v = -Y e_X + X e_Y + sin(X*Y*Z) e_Z
```

By default the LaTeX symbols of the coordinate coincide with the letters given within the angle brackets. But this can be adjusted through the optional argument `symbols` of the function `EuclideanSpace`, which has to be a string, usually prefixed by `r` (for raw string, in order to allow for the backslash character of LaTeX expressions). This string contains the coordinate fields separated by a blank space; each field contains the coordinate’s text symbol and possibly the coordinate’s LaTeX symbol (when the latter is different from the text symbol), both symbols being separated by a colon (:):

```sage
E.<xi,et,ze> = EuclideanSpace(symbols=r"xi:xi et:\eta ze:ze")
E.atlas()
[Chart (E^3, (xi, et, ze))]
v = E.vector_field(-et, xi, sin(xi*et*ze), name='v')
v.display()
v = -et e_xi + xi e_et + sin(et*xi*ze) e_ze
```

How to perform vector calculus in curvilinear coordinates

This tutorial introduces some vector calculus capabilities of SageMath within the 3-dimensional Euclidean space. The corresponding tools have been developed via the `SageManifolds` project.

The tutorial is also available as a Jupyter notebook, either passive (`nbviewer`) or interactive (`binder`).

Using spherical coordinates

To use spherical coordinates \((r, \theta, \phi)\) within the Euclidean 3-space \(\mathbb{E}^3\), it suffices to declare the latter with the keyword `coordinates='spherical'`:

```sage
E.<r,th,ph> = EuclideanSpace(coordinates='spherical')
E
Euclidean space E^3
```
Thanks to the notation \( \langle r, \theta, \phi \rangle \) in the above declaration, the coordinates \((r, \theta, \phi)\) are immediately available as three symbolic variables \(r, \theta, \phi\) (there is no need to declare them via \texttt{var()}, i.e. to type \( r, \theta, \phi = \texttt{var('r \ \theta \ \phi')} \)):

```python
sage: r is E.spherical_coordinates()[1]
True
sage: (r, th, ph) == E.spherical_coordinates()[:]
True
sage: type(r)
<class 'sage.symbolic.expression.Expression'>
```

Moreover, the coordinate LaTeX symbols are already set:

```python
sage: latex(th)
{\theta}
```

The coordinate ranges are:

```python
sage: E.spherical_coordinates().coord_range()
r: (0, +\infty); \theta: (0, \pi); \phi: [0, 2\pi] (periodic)
```

\( \mathbb{E}^3 \) is endowed with the \textit{orthonormal} vector frame \((e_r, e_\theta, e_\phi)\) associated with spherical coordinates:

```python
sage: E.frames()
[Coordinate frame (E^3, (\frac{\partial}{\partial r}, \frac{\partial}{\partial \theta}, \frac{\partial}{\partial \phi})),
 Vector frame (E^3, (e_r,e_\theta,e_\phi))]
```

In the above output, \( \left( \frac{\partial}{\partial r}, \frac{\partial}{\partial \theta}, \frac{\partial}{\partial \phi} \right) = \left( \frac{\partial}{\partial r}, \frac{\partial}{\partial \theta}, \frac{\partial}{\partial \phi} \right) \) is the \textit{coordinate} frame associated with \((r, \theta, \phi)\); it is not an orthonormal frame and will not be used below. The default frame is the orthonormal one:

```python
sage: E.default_frame()
Vector frame (E^3, (e_r,e_\theta,e_\phi))
```

### Defining a vector field

We define a vector field on \( \mathbb{E}^3 \) from its components in the orthonormal vector frame \((e_r, e_\theta, e_\phi)\):

```python
sage: v = E.vector_field(r*sin(2*ph)*sin(th)^2 + r,
....: r*cos(th)*sin(2*ph)*sin(th),
....: 2*r*cos(ph)^2*sin(th), name='v')
sage: v.display()
v = (r*sin(2*ph)*sin(th)^2 + r) e_r + r*cos(th)*sin(2*ph)*sin(th) e_th
+ 2*r*cos(ph)^2*sin(th) e_ph
```

We can access to the components of \( v \) via the square bracket operator:

```python
sage: v[1]
\( r*sin(2*ph)*sin(th)^2 + r \)
sage: v[:]
\[ r*sin(2*ph)*sin(th)^2 + r, r*cos(th)*sin(2*ph)*sin(th), 2*r*cos(ph)^2*sin(th) \]
```

A vector field can evaluated at any point of \( \mathbb{E}^3 \):
sage: p = E((1, pi/2, pi), name='p')
sage: p
Point p on the Euclidean space E^3
sage: p.coordinates()
(1, 1/2*pi, pi)
sage: vp = v.at(p)
sage: vp
Vector v at Point p on the Euclidean space E^3
sage: vp.display()
v = e_r + 2 e_ph

We may define a vector field with generic components:

sage: u = E.vector_field(function('u_r')(r,th,ph),
....: function('u_theta')(r,th,ph),
....: function('u_phi')(r,th,ph),
....: name='u')
sage: u.display()
u = u_r(r, th, ph) e_r + u_theta(r, th, ph) e_th + u_phi(r, th, ph) e_ph
sage: u[:]
[u_r(r, th, ph), u_theta(r, th, ph), u_phi(r, th, ph)]

Its value at the point \( p \) is then:

sage: up = u.at(p)
sage: up.display()
u = u_r(1, 1/2*pi, pi) e_r + u_theta(1, 1/2*pi, pi) e_th
+ u_phi(1, 1/2*pi, pi) e_ph

Differential operators in spherical coordinates

The standard operators \texttt{grad}, \texttt{div}, \texttt{curl}, etc. involved in vector calculus are accessible as methods on scalar fields and vector fields (e.g. \texttt{v.div()}). However, to allow for standard mathematical notations (e.g. \texttt{div(v)}), let us import the functions \texttt{grad()}, \texttt{div()}, \texttt{curl()} and \texttt{laplacian()}:

sage: from sage.manifolds.operators import *

Gradient

We first introduce a scalar field, via its expression in terms of Cartesian coordinates; in this example, we consider an unspecified function of \((r, \theta, \phi)\):

sage: F = E.scalar_field(function('f')(r,th,ph), name='F')
sage: F.display()
F: E^3 → \mathbb{R}
    (r, th, ph) ↦ f(r, th, ph)

The value of \( F \) at a point:

sage: F(p)
f(1, 1/2*pi, pi)
The gradient of $F$:

```
sage: grad(F)
Vector field grad(F) on the Euclidean space $E^3$
sage: grad(F).display()
grad(F) = $\frac{d(f)}{dr} e_r + \frac{d(f)}{d\theta}/r e_{\theta} + \frac{d(f)}{d\phi}/(r\sin(\theta)) e_{\phi}$
sage: norm(grad(F)).display()
|grad(F)|: $E^3 \to \mathbb{R}$(r, $\theta$, $\phi$)↦$\sqrt{\left(\frac{d(f)}{dr}^2 + \frac{d(f)}{d\theta}^2\right)\sin(\theta)^2 + \frac{d(f)}{d\phi}^2}/(r\sin(\theta))$
```

**Divergence**

The divergence of a vector field:

```
sage: s = div(u)
sage: s.display()
div(u): $E^3 \to \mathbb{R}$(r, $\theta$, $\phi$)↦$\left(\frac{r\frac{d(u_r)}{dr} + 2u_r(r, \theta, \phi) + \frac{d(u_\theta)}{d\theta}}{r\sin(\theta)} + \cos(\theta)\frac{u_\theta(r, \theta, \phi)}{r\sin(\theta)} + \frac{d(u_\phi)}{d\phi}/(r\sin(\theta))\right)$
sage: s.expr().expand()
2u_r(r, \theta, \phi)/r + \cos(\theta)u_\theta(r, \theta, \phi)/(r\sin(\theta)) + \frac{\partial u_\theta(r, \theta, \phi)}{\partial \theta}/r + \frac{\partial u_\phi(r, \theta, \phi)}{\partial \phi}/(r\sin(\theta)) + \frac{\partial u_r(r, \theta, \phi)}{\partial r}$
```

For $v$, we have:

```
sage: div(v).expr()
3
```

**Curl**

The curl of a vector field:

```
sage: s = curl(u)
sage: s
Vector field curl(u) on the Euclidean space $E^3$
sage: s.display()
curl(u) = $\frac{\cos(\theta)u_\phi(r, \theta, \phi) - \sin(\theta)d(u_\phi)/d\theta}{r\sin(\theta)} e_r - \frac{d(u_\theta)/d\phi}/(r\sin(\theta)) e_{\theta} - \frac{\partial d(u_\phi)/dr + u_\phi(r, \theta, \phi)}{r\sin(\theta)} e_{\phi}$
```

For $v$, we have:

```
sage: curl(v).display()
curl(v) = 2\cos(\theta) e_r - 2\sin(\theta) e_{\theta}$
```

The curl of a gradient is always zero:
**Laplacian**

The Laplacian of a scalar field:

```python
sage: s = laplacian(F)
sage: s.display()
```

\[
\Delta(F): E^3 \rightarrow \mathbb{R}
\]

\[
(r, \theta, \phi) \mapsto \frac{\left( r^2 \frac{d^2(f)}{dr^2} + 2r \frac{d(f)}{dr} + \frac{d^2(f)}{d\theta^2} \right) \sin^2(\theta) + \cos(\theta) \sin(\theta) \frac{d(f)}{d\theta}}{r^2 \sin^2(\theta)}
\]

\[
\Delta(F)^1 = \frac{\left( r^2 \frac{d^2(u_r)}{dr^2} + 2r \frac{d(u_r)}{dr} - 2u_r(r, \theta, \phi) + \frac{d^2(u_r)}{d\theta^2} \right) \sin^2(\theta)}{r^2 \sin^2(\theta)}
\]

\[
\Delta(F)^2 = \frac{\left( r^2 \frac{d^2(u_\theta)}{dr^2} + 2r \frac{d(u_\theta)}{dr} + 2\frac{d(u_r)}{d\theta} + \frac{d^2(u_\theta)}{d\theta^2} \right) \sin^2(\theta)}{r^2 \sin^2(\theta)}
\]

\[
\Delta(F)^3 = \frac{\left( r^2 \frac{d^2(u_\phi)}{dr^2} + 2r \frac{d(u_\phi)}{dr} + \frac{d^2(u_\phi)}{d\theta^2} \right) \sin^2(\theta) + \cos(\theta) \sin(\theta) \frac{d(u_\phi)}{d\theta}}{r^2 \sin^2(\theta)}
\]

The Laplacian of a vector field:

```python
sage: Du = laplacian(u)
sage: Du.display()
```

\[
\Delta(u) = \frac{\left( r^2 \frac{d^2(u_r)}{dr^2} + 2r \frac{d(u_r)}{dr} - 2u_r(r, \theta, \phi) + \frac{d^2(u_r)}{d\theta^2} \right) \sin^2(\theta) - \frac{2\frac{d(u_\theta)}{d\theta} \cos(\theta) + 2\frac{d(u_\phi)}{d\phi} \sin(\theta)}{r^2 \sin^2(\theta)} e_r}{e_r}
\]

Since this expression is quite lengthy, we may ask for a display component by component:

```python
sage: Du.display_comp()
```

\[
\Delta(u)^1 = \frac{\left( r^2 \frac{d^2(u_r)}{dr^2} + 2r \frac{d(u_r)}{dr} - 2u_r(r, \theta, \phi) + \frac{d^2(u_r)}{d\theta^2} \right) \sin^2(\theta)}{r^2 \sin^2(\theta)}
\]

\[
\Delta(u)^2 = \frac{\left( r^2 \frac{d^2(u_\theta)}{dr^2} + 2r \frac{d(u_\theta)}{dr} + 2\frac{d(u_r)}{d\theta} + \frac{d^2(u_\theta)}{d\theta^2} \right) \sin^2(\theta)}{r^2 \sin^2(\theta)}
\]

\[
\Delta(u)^3 = \frac{\left( r^2 \frac{d^2(u_\phi)}{dr^2} + 2r \frac{d(u_\phi)}{dr} + \frac{d^2(u_\phi)}{d\theta^2} \right) \sin^2(\theta) + \cos(\theta) \sin(\theta) \frac{d(u_\phi)}{d\theta}}{r^2 \sin^2(\theta)}
\]

We may expand each component:
As a test, we may check that these formulas coincide with those of Wikipedia's article Del in cylindrical and spherical coordinates.

### Using cylindrical coordinates

The use of cylindrical coordinates \((\rho, \phi, z)\) in the Euclidean space \(\mathbb{E}^3\) is on the same footing as that of spherical coordinates. To start with, one has simply to declare:

```python
sage: E.<rh, ph, z> = EuclideanSpace(coordinates='cylindrical')
```

The coordinate ranges are then:

```python
sage: E.cylindrical_coordinates().coord_range()
rh: \((0, +\infty)\); ph: \([0, 2\pi]\) (periodic); z: \((-\infty, +\infty)\)
```

The default vector frame is the orthonormal frame \((e_\rho, e_\phi, e_z)\) associated with cylindrical coordinates:

```python
sage: E.default_frame()
Vector frame (E^3, (e_rh,e_ph,e_z))
```

and one may define vector fields from their components in that frame:

```python
sage: v = E.vector_field(rh*(1+sin(2*ph)), 2*rh*cos(ph)^2, z, name='v')
sage: v.display()
v = rh*(\sin(2*\phi) + 1) e_rh + 2*rh*cos(\phi)^2 e_ph + z e_z
sage: v[:]
[rh*(\sin(2*\phi) + 1), 2*rh*cos(\phi)^2, z]
```

```python
sage: u = E.vector_field(function('u_rho')(rh, ph, z),
                    function('u_phi')(rh, ph, z),
                    function('u_z')(rh, ph, z),
                    name='u')
sage: u.display()
u = u_rho(rh, ph, z) e_rh + u_phi(rh, ph, z) e_ph + u_z(rh, ph, z) e_z
sage: u[:]
[u_rho(rh, ph, z), u_phi(rh, ph, z), u_z(rh, ph, z)]
```
Differential operators in cylindrical coordinates

```python
sage: from sage.manifolds.operators import *
```

The gradient:

```python
sage: F = E.scalar_field(function('f')(rh,ph,z), name='F')
sage: F.display()
F: E^3 \rightarrow \mathbb{R}
   (rh, ph, z) \mapsto f(rh, ph, z)
sage: grad(F)
Vector field grad(F) on the Euclidean space E^3
sage: grad(F).display()
grad(F) = \frac{d(f)}{drh} e_{rh} + \frac{d(f)}{dph/rh} e_{ph} + \frac{d(f)}{dz} e_z
```

The divergence:

```python
sage: s = div(u)
sage: s.display()
div(u): E^3 \rightarrow \mathbb{R}
   (rh, ph, z) \mapsto \frac{rh*d(u_rho)/drh + rh*d(u_z)/dz + u_rho(rh, ph, z) + d(u_phi)/dph}{rh}
sage: s.expr().expand()
\frac{u_rho(rh, ph, z)}{rh} + \frac{diff(u_phi(rh, ph, z), ph)}{rh} + \frac{diff(u_rho(rh, ph, z), rh)}{rh} + \frac{diff(u_z(rh, ph, z), z)}{rh}
```

The curl:

```python
sage: s = curl(u)
sage: s.display()
Vector field curl(u) on the Euclidean space E^3
sage: s.display()
curl(u) = \frac{-(rh*d(u_phi)/dz - d(u_z)/dph)}{rh} e_{rh} + \frac{rh*d(u_rho)/dz - d(u_z)/drh}{rh} e_{ph}
   + \frac{rh*d(u_phi)/drh + u_phi(rh, ph, z) - d(u_rho)/dph}{rh} e_z
```

The Laplacian of a scalar field:

```python
sage: s = laplacian(F)
sage: s.display()
Delta(F): E^3 \rightarrow \mathbb{R}
   (rh, ph, z) \mapsto \frac{rh^2*d^2(f)/drh^2 + rh^2*d^2(f)/dz^2 + rh*d(f)/drh}{rh^2}
   + \frac{d^2(f)/dph^2}{rh^2}
sage: s.expr().expand()
\frac{diff(f(rh, ph, z), rh)}{rh} + \frac{diff(f(rh, ph, z), ph, ph)}{rh^2}
   + \frac{diff(f(rh, ph, z), rh, rh)}{rh} + \frac{diff(f(rh, ph, z), z, z)}{rh}
```

The Laplacian of a vector field:

```python
sage: Du = laplacian(u)
sage: Du.display()
Delta(u) = \frac{rh^2*d^2(u_rho)/drh^2 + rh^2*d^2(u_rho)/dz^2 + rh*d(u_rho)/drh}{rh^2}
   - u_rho(rh, ph, z) - 2*d(u_phi)/dph + \frac{d^2(u_rho)/dph^2}{rh^2} e_{rh}
   + \frac{rh^2*d^2(u_phi)/drh^2 + rh^2*d^2(u_phi)/dz^2 + rh*d(u_phi)/drh}{rh^2}
   - u_phi(rh, ph, z) + \frac{d^2(u_phi)/dph^2 + 2*d(u_rho)/dph}{rh^2} e_{ph}
```

(continues on next page)
Again, we may check that the above formulas coincide with those of Wikipedia’s article Del in cylindrical and spherical coordinates.

### Changing coordinates

Given the expression of a vector field in a given coordinate system, SageMath can compute its expression in another coordinate system, see the tutorial *How to change coordinates*

### How to change coordinates

This tutorial introduces some vector calculus capabilities of SageMath within the 3-dimensional Euclidean space. The corresponding tools have been developed via the SageManifolds project.

The tutorial is also available as a Jupyter notebook, either passive ([nbviewer](https://nbviewer.jupyter.org)) or interactive ([binder](https://mybinder.org)).

### Starting from Cartesian coordinates

In this tutorial, we choose to start from the Cartesian coordinates \((x, y, z)\). Hence, we declare the 3-dimensional Euclidean space \(E^3\) as:

```python
sage: E.<x,y,z> = EuclideanSpace()
sage: E
Euclidean space E^3
```

By default, i.e. without the optional argument `coordinates` in `EuclideanSpace`, \(E^3\) is initialized with the chart of Cartesian coordinates:

```python
sage: E.atlas()
[Chart (E^3, (x, y, z))]
```

See the tutorial *How to perform vector calculus in curvilinear coordinates* for examples of initialization of the Euclidean space with spherical coordinates or cylindrical coordinates instead of the Cartesian ones.

Let us denote by `cartesian` the chart of Cartesian coordinates:

```python
sage: cartesian = E.cartesian_coordinates()
sage: cartesian
Chart (E^3, (x, y, z))
```
The access to the individual coordinates is performed via the square bracket operator:

```
sage: cartesian[1]
x
sage: cartesian[:]
(x, y, z)
```

Thanks to use of \(<x,y,z>\) when declaring \(E\), the Python variables \(x, y\) and \(z\) have been created to store the coordinates \((x, y, z)\) as symbolic expressions. There is no need to declare them via \(\text{var}()\), i.e. to type \(x, y, z = \text{var}('x y z')\); they are immediately available:

```
sage: y is cartesian[2]
True
sage: type(y)
<class 'sage.symbolic.expression.Expression'>
```

Each of the Cartesian coordinates spans the entire real line:

```
sage: cartesian.coord_range()
x: (-oo, +oo); y: (-oo, +oo); z: (-oo, +oo)
```

Being the only coordinate chart created so far, \(\text{cartesian}\) is the default chart on \(E\):

```
sage: cartesian is E.default_chart()
True
```

\(\mathbb{R}^3\) is endowed with the orthonormal vector frame \((e_x, e_y, e_z)\) associated with Cartesian coordinates:

```
sage: E.frames()
[Coordinate frame \((E^3, (e_x,e_y,e_z))\)]
```

Let us denote it by \(\text{cartesian\_frame}\):

```
sage: cartesian_frame = E.cartesian_frame()
sage: cartesian_frame
Coordinate frame \((E^3, (e_x,e_y,e_z))\)
sage: cartesian_frame is E.default_frame()
True
```

Each element of this frame is a unit vector field; for instance, we have \(e_x \cdot e_x = 1\):

```
sage: e_x = cartesian_frame[1]
sage: e_x
Vector field e_x on the Euclidean space E^3
sage: e_x.dot(e_x).expr()
1
```

as well as \(e_x \cdot e_y = 0\):

```
sage: e_y = cartesian_frame[2]
sage: e_x.dot(e_y).expr()
0
```
Introducing spherical coordinates

Spherical coordinates are introduced by:

```sage
spherical.<r,th,ph> = E.spherical_coordinates()
spherical
Chart (E^3, (r, th, ph))
```

We have:

```sage
spherical[:]
(r, th, ph)
spherical.coord_range()
r: (0, +oo); th: (0, pi); ph: [0, 2*pi] (periodic)
```

$E^3$ is now endowed with two coordinate charts:

```sage
E.atlas()
[Chart (E^3, (x, y, z)), Chart (E^3, (r, th, ph))]
```

The change-of-coordinate formulas have been automatically implemented during the above call `E.spherical_coordinates()`:

```sage
E.coord_change(spherical, cartesian).display()
x = r*cos(ph)*sin(th)
y = r*sin(ph)*sin(th)
z = r*cos(th)
sage: E.coord_change(cartesian, spherical).display()
r = sqrt(x^2 + y^2 + z^2)
th = arctan2(sqrt(x^2 + y^2), z)
ph = arctan2(y, x)
```

These formulas are automatically used if we ask to plot the grid of spherical coordinates in terms of Cartesian coordinates:

```sage
spherical.plot(cartesian, color={r:'red', th:'green', ph:'orange'})
```

Graphics3d Object

Note that

- the red lines are those along which $r$ varies, while $(\theta, \phi)$ are kept fixed;
- the grid lines are those along which $\theta$ varies, while $(r, \phi)$ are kept fixed;
- the orange lines are those along which $\phi$ varies, while $(r, \theta)$ are kept fixed.

For customizing the plot, see the list of options in the documentation of `plot()`. For instance, we may draw the spherical coordinates in the plane $\theta = \pi/2$ in terms of the coordinates $(x, y)$:

```sage
spherical.plot(cartesian, fixed_coords={th: pi/2}, ambient_coords=(x,y),
....:          color={r:'red', th:'green', ph:'orange'})
```

Graphics object consisting of 18 graphics primitives

Similarly the grid of spherical coordinates in the half-plane $\phi = 0$ drawn in terms of the coordinates $(x, z)$ is obtained via:
12.1. Thematic tutorial document tree
Relations between the Cartesian and spherical vector frames

At this stage, \( \mathbb{E}^3 \) is endowed with three vector frames:

\[
\text{sage: } E \text{.frames()}
\]

\[
\text{[Coordinate frame (E^3, (e_x,e_y,e_z)), Coordinate frame (E^3, (\partial/\partial r,\partial/\partial \theta,\partial/\partial \phi)), Vector frame (E^3, (e_r,e_th,e_ph))]} 
\]

The second one is the coordinate frame \( \left( \frac{\partial}{\partial r}, \frac{\partial}{\partial \theta}, \frac{\partial}{\partial \phi} \right) \) of spherical coordinates, while the third one is the standard orthonormal frame \( (e_r, e_\theta, e_\phi) \) associated with spherical coordinates. For Cartesian coordinates, the coordinate frame and the orthonormal frame coincide: it is \( (e_x, e_y, e_z) \). For spherical coordinates, the orthonormal frame is returned by the method \text{spherical\_frame()}:

\[
\text{sage: spherical\_frame = E\text{.spherical\_frame()}}
\]
\[
\text{sage: spherical\_frame}
\]

\[
\text{Vector frame (E^3, (e_r,e_th,e_ph))} 
\]
We may check that it is an orthonormal frame, i.e. that it obeys $e_i \cdot e_j = \delta_{ij}$:

```python
sage: es = spherical_frame
sage: [[es[i].dot(es[j]).expr() for j in E.irange()] for i in E.irange()]
[[1, 0, 0], [0, 1, 0], [0, 0, 1]]
```

Via the method display, we may express the orthonormal spherical frame in terms of the Cartesian one:

```python
sage: for vec in spherical_frame:
....:     vec.display(cartesian_frame, spherical)
```

\[
\begin{align*}
e_r &= \cos(\phi)\sin(\theta) e_x + \sin(\phi)\sin(\theta) e_y + \cos(\theta) e_z \\
e_{\theta} &= \cos(\phi)\cos(\theta) e_x + \cos(\theta)\sin(\phi) e_y - \sin(\theta) e_z \\
e_{\phi} &= -\sin(\phi) e_x + \cos(\phi) e_y
\end{align*}
\]

The reverse is:

```python
sage: for vec in cartesian_frame:
....:     vec.display(spherical_frame, spherical)
```

\[e_x = \cos(\phi)\sin(\theta) e_r + \cos(\phi)\cos(\theta) e_{\theta} - \sin(\phi) e_{\phi}
\]

\[e_y = \sin(\phi)\sin(\theta) e_r + \cos(\theta)\sin(\phi) e_{\theta} + \cos(\phi) e_{\phi}
\]

\[e_z = \cos(\theta) e_r - \sin(\theta) e_{\theta}
\]

We may also express the orthonormal frame $\left(e_r, e_{\theta}, e_{\phi}\right)$ in terms on the coordinate frame \(\left(\frac{\partial}{\partial r}, \frac{\partial}{\partial \theta}, \frac{\partial}{\partial \phi}\right)\) (the latter being returned by the method `frame()` acting on the chart `spherical`):

```python
sage: for vec in spherical_frame:
....:     vec.display(spherical.frame(), spherical)
```

\[e_r = \frac{\partial}{\partial r}
\]

\[e_{\theta} = \frac{1}{r} \frac{\partial}{\partial \theta}
\]

\[e_{\phi} = \frac{1}{r \sin(\theta)} \frac{\partial}{\partial \phi}
\]

### Introducing cylindrical coordinates

Cylindrical coordinates are introduced in a way similar to spherical coordinates:

```python
sage: cylindrical.<rh,ph,z> = E.cylindrical_coordinates()
sage: cylindrical
Chart (E^3, (rh, ph, z))
```

We have:

```python
sage: cylindrical[:]
(rh, ph, z)
sage: rh is cylindrical[1]
True
```

\[\rho: (0, +\infty); \phi: [0, 2\pi] \text{ (periodic)}; z: (-\infty, +\infty)\]

$E^3$ is now endowed with three coordinate charts:

```python
sage: E.atlas()
[Chart (E^3, (x, y, z)), Chart (E^3, (r, th, ph)), Chart (E^3, (rh, ph, z))]
```

The transformations linking the cylindrical coordinates to the Cartesian ones are:
```
sage: E.coord_change(cylindrical, cartesian).display()
x = rh*cos(ph)
y = rh*sin(ph)
z = z
sage: E.coord_change(cartesian, cylindrical).display()
rh = sqrt(x^2 + y^2)
ph = arctan2(y, x)
z = z
```

There are now five vector frames defined on $E^3$:

```
sage: E.frames()
[Coordinate frame (E^3, (e_x,e_y,e_z)),
 Coordinate frame (E^3, (∂/∂r,∂/∂θ,∂/∂φ)),
 Vector frame (E^3, (e_r,e_th,e_ph)),
 Coordinate frame (E^3, (∂/∂rh,∂/∂ph,∂/∂z)),
 Vector frame (E^3, (e_rh,e_ph,e_z))]
```

The orthonormal frame associated with cylindrical coordinates is $(e_ρ, e_φ, e_z)$:

```
sage: cylindrical_frame = E.cylindrical_frame()
sage: cylindrical_frame
Vector frame (E^3, (e_rh,e_ph,e_z))
```

We may check that it is an orthonormal frame:

```
sage: ec = cylindrical_frame
sage: [[ec[i].dot(ec[j]).expr() for j in E.irange()] for i in E.irange()]
[[1, 0, 0], [0, 1, 0], [0, 0, 1]]
```

and express it in terms of the Cartesian frame:

```
sage: for vec in cylindrical_frame:
    ....:    vec.display(cartesian_frame, cylindrical)
e_rh = cos(ph) e_x + sin(ph) e_y
e_ph = -sin(ph) e_x + cos(ph) e_y
e_z = e_z
```

The reverse is:

```
sage: for vec in cartesian_frame:
    ....:    vec.display(cylindrical_frame, cylindrical)
e_x = cos(ph) e_rh - sin(ph) e_ph
e_y = sin(ph) e_rh + cos(ph) e_ph
e_z = e_z
```

Of course, we may express the orthonormal cylindrical frame in terms of the spherical one:

```
sage: for vec in cylindrical_frame:
    ....:    vec.display(spherical_frame, spherical)
e_rh = sin(th) e_r + cos(th) e_th
e_ph = e_ph
e_z = cos(th) e_r - sin(th) e_th
```

along with the reverse transformation:
The orthonormal frame \((e_\rho, e_\phi, e_z)\) can be expressed in terms of the coordinate frame \(\left(\frac{\partial}{\partial \rho}, \frac{\partial}{\partial \phi}, \frac{\partial}{\partial z}\right)\) (the latter being returned by the method \texttt{frame()} acting on the chart \texttt{cylindrical}):
How to express a scalar field in various coordinate systems

Let us define a scalar field on $\mathbb{R}^3$ from its expression in Cartesian coordinates:

```
sage: f = E.scalar_field(x^2+y^2 - z^2, name='f')
```

Note that since the Cartesian coordinates are the default ones, we have not specified them in the above definition. Thanks to the known coordinate transformations, the expression of $f$ in terms of other coordinates is automatically computed:

```
sage: f.display()
f: $E^3 \to \mathbb{R}$
    (x, y, z) ↦ $x^2 + y^2 - z^2$
    (r, th, ph) ↦ $-2*r^2*cos(th)^2 + r^2$
    (rh, ph, z) ↦ $rh^2 - z^2$
```

We can limit the output to a single coordinate system:

```
sage: f.display(cartesian)
f: $E^3 \to \mathbb{R}$
    (x, y, z) ↦ $x^2 + y^2 - z^2$
sage: f.display(cylindrical)
f: $E^3 \to \mathbb{R}$
    (rh, ph, z) ↦ $rh^2 - z^2$
```

The coordinate expression in a given coordinate system is obtained via the method `expr()`:

```
sage: f.expr()  # expression in the default chart (Cartesian coordinates)
x^2 + y^2 - z^2
sage: f.expr(spherical)
-2*r^2*cos(th)^2 + r^2
sage: f.expr(cylindrical)
rh^2 - z^2
```

The values of $f$ at points $p$ and $q$ are:

```
sage: f(p)
2
sage: f(q)
8
```

Of course, we may define a scalar field from its coordinate expression in a chart that is not the default one:

```
sage: g = E.scalar_field(r^2, chart=spherical, name='g')
```

Instead of using the keyword argument `chart`, one can pass a dictionary as the first argument, with the chart as key:

```
sage: g = E.scalar_field({spherical: r^2}, name='g')
```

The computation of the expressions of $g$ in the other coordinate systems is triggered by the method `display()`:

```
sage: g.display()
g: $E^3 \to \mathbb{R}$
    (x, y, z) ↦ $x^2 + y^2 + z^2$
```

(continues on next page)
How to express a vector field in various frames

Let us introduce a vector field on $\mathbb{R}^3$ by its components in the Cartesian frame. Since the latter is the default vector frame on $\mathbb{R}^3$, it suffices to write:

```sage
sage: v = E.vector_field(-y, x, z^2, name='v')
sage: v.display()
v = -y e_x + x e_y + z^2 e_z
```

Equivalently, a vector field can be defined directly from its expansion on the Cartesian frame:

```sage
sage: ex, ey, ez = cartesian_frame[:]
sage: v = -y*ex + x*ey + z^2*ez
sage: v.display()
-y e_x + x e_y + z^2 e_z
```

Let us provide $v$ with some name, as above:

```sage
sage: v.set_name('v')
sage: v.display()
v = -y e_x + x e_y + z^2 e_z
```

The components of $v$ are returned by the square bracket operator:

```sage
sage: v[1]
-y
sage: v[:]
[-y, x, z^2]
```

The computation of the expression of $v$ in terms of the orthonormal spherical frame is triggered by the method display():
Thematic Tutorials, Release 9.7

\[
\text{sage: } v[\text{spherical\_frame, :, spherical}]
\]
\[
[r^2\cos(\text{th})^3, -r^2\cos(\text{th})^2\sin(\text{th}), r\sin(\text{th})]
\]

Similarly, the expression of \( v \) in terms of the cylindrical frame is:

\[
\text{sage: } v.\text{display(cylindrical\_frame, cylindrical)}
\]
\[
v = rh \ e_\text{ph} + z^2 \ e_z
\]
\[
\text{sage: } v[\text{cylindrical\_frame, :, cylindrical}]
\]
\[
[0, rh, z^2]
\]

The value of the vector field \( v \) at point \( p \) is:

\[
\text{sage: } \text{vp} = v.\text{at(p)}
\]
\[
\text{sage: } \text{vp}
\]
\[
\text{Vector \( v \) at Point \( p \) on the Euclidean space \( E^3 \)}
\]
\[
\text{sage: } \text{vp.}\text{display()}
\]
\[
v = -e_x - e_y
\]
\[
\text{sage: } \text{vp.}\text{display(spherical\_frame.}\text{at(p)})
\]
\[
v = \sqrt{2} \ e_\text{ph}
\]
\[
\text{sage: } \text{vp.}\text{display(cylindrical\_frame.}\text{at(p)})
\]
\[
v = \sqrt{2} \ e_\text{ph}
\]

The value of the vector field \( v \) at point \( q \) is:

\[
\text{sage: } \text{vq} = v.\text{at(q)}
\]
\[
\text{sage: } \text{vq}
\]
\[
\text{Vector \( v \) at Point \( q \) on the Euclidean space \( E^3 \)}
\]
\[
\text{sage: } \text{vq.}\text{display()}
\]
\[
v = -2\sqrt{3} \ e_y + 4 \ e_z
\]
\[
\text{sage: } \text{vq.}\text{display(spherical\_frame.}\text{at(q)})
\]
\[
v = 2 \ e_r - 2\sqrt{3} \ e_\text{th} + 2\sqrt{3} \ e_\text{ph}
\]
\[
\text{sage: } \text{vq.}\text{display(cylindrical\_frame.}\text{at(q)})
\]
\[
v = 2\sqrt{3} \ e_\text{ph} + 4 \ e_z
\]

**How to change the default coordinates and default vector frame**

At any time, one may change the default coordinates by the method `set_default_chart()`:

\[
\text{sage: } E.\text{set_default_chart(spherical)}
\]

Then:

\[
\text{sage: } f.\text{expr()}
\]
\[
-2\sqrt{3} \sqrt{2} \cos(\text{th})^2 + r^2
\]
\[
\text{sage: } v.\text{display()}
\]
\[
v = -r\sin(\text{ph})\sin(\text{th}) \ e_\text{x} + r\cos(\text{ph})\sin(\text{th}) \ e_\text{y} + r\sqrt{2}\cos(\text{th})^2 \ e_z
\]

Note that the default vector frame is still the Cartesian one; to change to the orthonormal spherical frame, use `set_default_frame()`:

\[
\text{sage: } E.\text{set_default_frame(spherical\_frame)}
\]

Then:
Advanced aspects: the Euclidean space as a Riemannian manifold

This tutorial introduces some vector calculus capabilities of SageMath within the 3-dimensional Euclidean space. The corresponding tools have been developed via the SageManifolds project.

The tutorial is also available as a Jupyter notebook, either passive (nbviewer) or interactive (binder).

The Euclidean 3-space

Let us consider the 3-dimensional Euclidean space \( \mathbb{E}^3 \), with Cartesian coordinates \((x, y, z)\):

```sage
E.<x,y,z> = EuclideanSpace()
E
```

\( \mathbb{E}^3 \) is actually a Riemannian manifold (see pseudo_riemannian), i.e. a smooth real manifold endowed with a positive definite metric tensor:

```sage
E.category()

Join of
Category of smooth manifolds over Real Field with 53 bits of precision and
Category of connected manifolds over Real Field with 53 bits of precision and
Category of complete metric spaces

E.base_field() is RR
True

E.metric()
Riemannian metric g on the Euclidean space \( \mathbb{E}^3 \)
```

Actually RR is used here as a proxy for the real field (this should be replaced in the future, see the discussion at trac ticket #24456) and the 53 bits of precision play of course no role for the symbolic computations.

Let us introduce spherical and cylindrical coordinates on \( \mathbb{E}^3 \):

```sage
spherical.<r,th,ph> = E.spherical_coordinates()

sage: cylindrical.<rh,ph,z> = E.cylindrical_coordinates()
```

The user atlas of \( \mathbb{E}^3 \) has then three charts:

```sage
E.atlas()

[Chart (\( \mathbb{E}^3 \), (x, y, z)), Chart (\( \mathbb{E}^3 \), (r, th, ph)), Chart (\( \mathbb{E}^3 \), (rh, ph, z))]
```

while there are five vector frames defined on \( \mathbb{E}^3 \):

```sage
E.frames()

[Coordinate frame (\( \mathbb{E}^3 \), (e_x,e_y,e_z)),
 Coordinate frame (\( \mathbb{E}^3 \), (\partial/\partial r,\partial/\partial th,\partial/\partial ph)),
 Vector frame (\( \mathbb{E}^3 \), (e_r,e_th,e_ph))]
```

(continues on next page)
Indeed, there are two frames associated with each of the three coordinate systems: the coordinate frame (denoted with partial derivatives above) and an orthonormal frame (denoted by \( e_\ast \) above), but for Cartesian coordinates, both frames coincide.

We get the orthonormal spherical and cylindrical frames by:

```
 sage: spherical_frame = E.spherical_frame()
sage: spherical_frame
Vector frame (E^3, (e_r,e_th,e_ph))
```

```
 sage: cylindrical_frame = E.cylindrical_frame()
sage: cylindrical_frame
Vector frame (E^3, (e_rh,e_ph,e_z))
```

On the other side, the coordinate frames \( \left( \frac{\partial}{\partial r}, \frac{\partial}{\partial \theta}, \frac{\partial}{\partial \phi} \right) \) and \( \left( \frac{\partial}{\partial \rho}, \frac{\partial}{\partial \phi}, \frac{\partial}{\partial z} \right) \) are returned by the method \texttt{frame()} acting on the coordinate charts:

```
 sage: spherical.frame()
Coordinate frame (E^3, (\frac{\partial}{\partial r}, \frac{\partial}{\partial \theta}, \frac{\partial}{\partial \phi}))
```

```
 sage: cylindrical.frame()
Coordinate frame (E^3, (\frac{\partial}{\partial rh}, \frac{\partial}{\partial ph}, \frac{\partial}{\partial z}))
```

**Charts as maps** \( \mathbb{E}^3 \rightarrow \mathbb{R}^3 \)

The chart of Cartesian coordinates has been constructed at the declaration of \( E \); let us denote it by \texttt{cartesian}:

```
 sage: cartesian = E.cartesian_coordinates()
sage: cartesian
Chart (E^3, (x, y, z))
```

Let us consider a point \( p \in \mathbb{E}^3 \), defined by its Cartesian coordinates:

```
 sage: p = E((-1, 1,0), chart=cartesian, name='p')
sage: p
Point p on the Euclidean space E^3
```

The coordinates of \( p \) in a given coordinate chart are obtained by letting the corresponding chart act on \( p \):

```
 sage: cartesian(p)
(-1, 1, 0)
sage: spherical(p)
(sqrt(2), 1/2*pi, 3/4*pi)
sage: cylindrical(p)
(sqrt(2), 3/4*pi, 0)
```
Riemannian metric

The default metric tensor of $E^3$ is:

```
sage: g = E.metric()
sage: g
Riemannian metric g on the Euclidean space $E^3$
sage: g.display()
g = dx⊗dx + dy⊗dy + dz⊗dz
sage: g[:]
[1 0 0]
[0 1 0]
[0 0 1]
```

The above display is performed in the default frame, which is the Cartesian one. Of course, we may ask for display with respect to other frames:

```
sage: g.display(spherical_frame)
g = e^r⊗e^r + e^th⊗e^th + e^ph⊗e^ph
sage: g[spherical_frame, :]
[1 0 0]
[0 1 0]
[0 0 1]
```

In the above display, $e^r = e^r$, $e^th = e^\theta$ and $e^ph = e^\phi$ are the 1-forms defining the coframe dual to the orthonormal spherical frame $(e_r, e_\theta, e_\phi)$:

```
sage: spherical_frame.coframe()
Coframe ($E^3$, $(e^r, e^\theta, e^\phi)$)
```

The fact that the above metric components are either 0 or 1 reflect the orthonormality of the vector frame $(e_r, e_\theta, e_\phi)$. On the contrary, in the coordinate frame $(\frac{\partial}{\partial r}, \frac{\partial}{\partial \theta}, \frac{\partial}{\partial \phi})$, which is not orthonormal, some components differ from 0 or 1:

```
sage: g.display(spherical.frame())
g = dr⊗dr + (x^2 + y^2 + z^2) dth⊗dth + (x^2 + y^2) dph⊗dph
```

Note that the components are expressed in terms of the default chart, namely the Cartesian one. To have them displayed in terms of the spherical chart, we have to provide the latter as the second argument of the method `display()`:

```
sage: g.display(spherical.frame(), spherical)
g = dr⊗dr + r^2 dth⊗dth + r^2*sin(th)^2 dph⊗dph
```

Since SageMath 8.8, a shortcut is:

```
sage: g.display(spherical)
g = dr⊗dr + r^2 dth⊗dth + r^2*sin(th)^2 dph⊗dph
```

The matrix view of the components is obtained via the square bracket operator:

```
sage: g[spherical.frame(), :, spherical]
[ 1 0 0]
[ 0 r^2 0]
[ 0 0 r^2*sin(th)^2]
```
Similarly, for cylindrical coordinates, we have:

```plaintext
sage: g.display(cylindrical_frame)
g = e^rh⊗e^rh + e^ph⊗e^ph + e^z⊗e^z
sage: g.display(cylindrical)
g = drh⊗drh + rh^2 dph⊗dph + dz⊗dz
sage: g[cylindrical.frame(), :, cylindrical]
[[ 1  0  0]
 [ 0 rh^2  0]
 [ 0  0   1]]
```

The metric $g$ is a flat: its Riemann curvature tensor (see `riemann()`) is zero:

```plaintext
sage: g.riemann()
Tensor field Riem(g) of type (1,3) on the Euclidean space E^3
sage: g.riemann().display()
Riem(g) = 0
```

The metric $g$ defines the dot product on $\mathbb{E}^3$:

```plaintext
sage: u = E.vector_field(x*y, y*z, z*x)
sage: u.display()
x*y e_x + y*z e_y + z*x e_z
sage: v = E.vector_field(-y, x, z^2, name='v')
sage: v.display()
v = -y e_x + x e_y + z^2 e_z
sage: u.dot(v) == g(u,v)
True
```

Consequently:

```plaintext
sage: norm(u) == sqrt(g(u,u))
True
```

### The Levi-Civita tensor

The scalar triple product of $\mathbb{E}^3$ is provided by the Levi-Civita tensor (also called *volume form*) associated with $g$ (and chosen such that $(e_x, e_y, e_z)$ is right-handed):

```plaintext
sage: epsilon = E.scalar_triple_product()
sage: epsilon
3-form epsilon on the Euclidean space E^3
sage: epsilon is E.volume_form()
True
sage: epsilon.display()
epsilon = dx∧dy∧dz
sage: epsilon.display(spherical)
epsilon = r^2*sin(th) dr∧dth∧dph
sage: epsilon.display(cylindrical)
epsilon = rh drh∧dph∧dz
```

Checking that all orthonormal frames introduced above are right-handed:
Vector fields as derivations

Let $f$ be a scalar field on $\mathbb{E}^3$:

```sage
f = E.scalar_field(x^2 + y^2 - z^2, name='f')
f.display()
```

Vector fields act as derivations on scalar fields:

```sage
v(f) == v.dot(f.gradient())
```

```sage
df = f.differential()
df.display()
v(f) == df(v)
```

12.1. Thematic tutorial document tree
The algebra of scalar fields

The set $C^\infty(E^3)$ of all smooth scalar fields on $E^3$ forms a commutative algebra over $\mathbb{R}$:

```python
sage: CE = E.scalar_field_algebra()
sage: CE
Algebra of differentiable scalar fields on the Euclidean space $E^3$
sage: CE.category()
Join of Category of commutative algebras over Symbolic Ring and Category of homsets of...
sage: f in CE
True
```

In SageMath terminology, $C^\infty(E^3)$ is the parent of scalar fields:

```python
sage: f.parent() is CE
True
```

The free module of vector fields

The set $\mathcal{X}(E^3)$ of all vector fields on $E^3$ is a free module of rank 3 over the commutative algebra $C^\infty(E^3)$:

```python
sage: XE = v.parent()
sage: XE
Free module $X(E^3)$ of vector fields on the Euclidean space $E^3$
sage: XE.category()
Category of finite dimensional modules over Algebra of differentiable scalar fields on the Euclidean space $E^3$
sage: XE.base_ring()
Algebra of differentiable scalar fields on the Euclidean space $E^3$
sage: XE.base_ring() is CE
True
sage: rank(XE)
3
```

The bases of the free module $\mathcal{X}(E^3)$ are nothing but the vector frames defined on $E^3$:

```python
sage: XE.bases()
[Coordinate frame $(E^3, (e_x,e_y,e_z))$, Coordinate frame $(E^3, (\partial/\partial r,\partial/\partial \theta,\partial/\partial \phi))$, Vector frame $(E^3, (e_r,e_{\theta},e_{\phi}))$, Coordinate frame $(E^3, (\partial/\partial r_{\phi},\partial/\partial \phi_{\theta},\partial/\partial z))$, Vector frame $(E^3, (e_{r_{\phi}},e_{\theta_{\phi}},e_{z}))$]```
Tangent spaces

A vector field evaluated at a point $p$ is a vector in the tangent space $T_p\mathbb{E}^3$:

```
sage: p
Point p on the Euclidean space $\mathbb{E}^3$
sage: vp = v.at(p)
sage: vp
Vector v at Point p on the Euclidean space $\mathbb{E}^3$
sage: vp.display()
v = -e_x - e_y
```

```
sage: Tp = vp.parent()
sage: Tp
Tangent space at Point p on the Euclidean space $\mathbb{E}^3$
sage: Tp is E.tangent_space(p)
True
sage: Tp.category()
Category of finite dimensional vector spaces over Symbolic Ring
dsage: dim(Tp)
3
sage: isinstance(Tp, FiniteRankFreeModule)
True
```

The bases on $T_p\mathbb{E}^3$ are inherited from the vector frames of $\mathbb{E}^3$:

```
sage: Tp.bases()
[Basis (e_x,e_y,e_z) on the Tangent space at Point p on the Euclidean space $\mathbb{E}^3$,
 Basis (∂/∂r,∂/∂th,∂/∂ph) on the Tangent space at Point p on the Euclidean space $\mathbb{E}^3$,
 Basis (e_r,e_th,e_ph) on the Tangent space at Point p on the Euclidean space $\mathbb{E}^3$,
 Basis (∂/∂rh,∂/∂ph,∂/∂z) on the Tangent space at Point p on the Euclidean space $\mathbb{E}^3$,
 Basis (e_rh,e_ph,e_z) on the Tangent space at Point p on the Euclidean space $\mathbb{E}^3$]
```

For instance, we have:

```
sage: spherical_frame.at(p)
Basis (e_r,e_th,e_ph) on the Tangent space at Point p on the Euclidean space $\mathbb{E}^3$
sage: spherical_frame.at(p) in Tp.bases()
True
```

Levi-Civita connection

The Levi-Civita connection associated to the Euclidean metric $g$ is:

```
sage: nabla = g.connection()
sage: nabla
Levi-Civita connection nabla_g associated with the Riemannian metric g
on the Euclidean space $\mathbb{E}^3$
```

The corresponding Christoffel symbols with respect to Cartesian coordinates are identically zero: none of them appear in the output of `christoffel_symbols_display()`, which by default displays only nonzero Christoffel symbols:
sage: g.christoffel_symbols_display(cartesian)

On the contrary, some of the Christoffel symbols with respect to spherical coordinates differ from zero:

sage: g.christoffel_symbols_display(spherical)
Gam^r_th,th = -r
Gam^r_ph,ph = -r*sin(th)^2
Gam^th_r,th = 1/r
Gam^th_ph,ph = -cos(th)*sin(th)
Gam^ph_r,ph = 1/r
Gam^ph_th,ph = cos(th)/sin(th)

By default, only nonzero and nonredundant values are displayed (for instance $\Gamma^\varphi_{\varphi r}$ is skipped, since it can be deduced from $\Gamma^\varphi_{r \varphi}$ by symmetry on the last two indices).
Similarly, the nonzero Christoffel symbols with respect to cylindrical coordinates are:

sage: g.christoffel_symbols_display(cylindrical)
Gam^rh_ph,ph = -rh
Gam^ph_rh,ph = 1/rh

The Christoffel symbols are nothing but the connection coefficients in the corresponding coordinate frame:

sage: nabla.display(cylindrical.frame(), cylindrical, only_nonredundant=True)
Gam^rh_ph,ph = -rh
Gam^ph_rh,ph = 1/rh

The connection coefficients with respect to the orthonormal (non-coordinate) frames are (again only nonzero values are displayed):

sage: nabla.display(spherical_frame, spherical)
Gam^1_22 = -1/r
Gam^1_33 = -1/r
Gam^2_12 = 1/r
Gam^2_33 = -cos(th)/(r*sin(th))
Gam^3_13 = 1/r
Gam^3_23 = cos(th)/(r*sin(th))
sage: nabla.display(cylindrical_frame, cylindrical)
Gam^1_22 = -1/rh
Gam^2_12 = 1/rh

The Levi-Civita connection $\nabla_g$ is the connection involved in the standard differential operators:

sage: from sage.manifolds.operators import *
sage: grad(f) == nabla(f).up(g)
True
sage: nabla(f) == grad(f).down(g)
True
sage: div(u) == nabla(u).trace()
True
sage: div(v) == nabla(v).trace()
True
sage: laplacian(f) == nabla(nabla(f).up(g)).trace()
True

(continues on next page)
Vector calculus in the Euclidean plane

This tutorial introduces some vector calculus capabilities of SageMath in the framework of the 2-dimensional Euclidean space. The corresponding tools have been developed via the SageManifolds project.

The tutorial is also available as a Jupyter notebook, either passive (nbviewer) or interactive (binder).

1. Defining the Euclidean plane

We define the Euclidean plane $\mathbb{E}^2$ as a 2-dimensional Euclidean space, with Cartesian coordinates $(x, y)$, via the function $\text{EuclideanSpace}$:

```python
sage: E.<x,y> = EuclideanSpace()
sage: E
Euclidean plane $\mathbb{E}^2$
```

Thanks to the use of $\langle x, y \rangle$ in the above command, the Python variables $x$ and $y$ are assigned to the symbolic variables $x$ and $y$ describing the Cartesian coordinates (there is no need to declare them via $\text{var}$, i.e. to type $x, y = \text{var}('x y')$):

```python
sage: type(y)
<class 'sage.symbolic.expression.Expression'>
```

Instead of using the variables $x$ and $y$, one may also access to the coordinates by their indices in the chart of Cartesian coordinates:

```python
sage: cartesian = E.cartesian_coordinates()
sage: cartesian
Chart (E^2, (x, y))
```

```python
sage: cartesian[1]
x
sage: cartesian[2]
y
sage: y is cartesian[2]
True
```

Each of the Cartesian coordinates spans the entire real line:

```python
sage: cartesian.coord_range()
x: (-\infty, +\infty); y: (-\infty, +\infty)
```
2. Vector fields

The Euclidean plane $\mathbb{E}^2$ is canonically endowed with the vector frame associated with Cartesian coordinates:

```
sage: E.default_frame()
Coordinate frame (E^2, (e_x,e_y))
```

Vector fields on $\mathbb{E}^2$ are then defined from their components in that frame:

```
sage: v = E.vector_field(-y, x, name='v')
sage: v.display()
v = -y e_x + x e_y
```

The access to individual components is performed by the square bracket operator:

```
sage: v[1]
-y
sage: v[:]
[-y, x]
```

A plot of the vector field $v$ (this is with the default parameters, see the documentation of `plot()` for the various options):

```
sage: v.plot()
Graphics object consisting of 80 graphics primitives
```
One may also define a vector field by setting the components in a second stage:

```python
sage: w = E.vector_field(name='w')
sage: w[1] = function('w_x')(x,y)
sage: w[2] = function('w_y')(x,y)
sage: w.display()
\[ w = w_x(x, y) \ e_x + w_y(x, y) \ e_y \]
```

Note that in the above example the components of \( w \) are unspecified functions of \((x, y)\), contrary to the components of \( v \).

Standard linear algebra operations can be performed on vector fields:

```python
sage: s = 2*v + x*w
sage: s.display()
\( (x \ w_x(x, y) - 2y) \ e_x + (x \ w_y(x, y) + 2x) \ e_y \)
```

### Scalar product and norm

The dot (or scalar) product \( u \cdot v \) of the vector fields \( u \) and \( v \) is obtained by the operator `dot_product()`; it gives rise to a scalar field on \( \mathbb{E}^2 \):

```python
sage: s = v.dot_product(w)
sage: s
Scalar field v.w on the Euclidean plane \( \mathbb{E}^2 \)
```

A shortcut alias of `dot_product()` is `dot`:

```python
sage: s == v.dot(w)
True
```

```python
sage: s.display()
\( v.w: \mathbb{E}^2 \rightarrow \mathbb{R} \)
\( (x, y) \mapsto -y w_x(x, y) + x w_y(x, y) \)
```

The symbolic expression representing the scalar field \( v \cdot w \) is obtained by means of the method `expr()`:

```python
sage: s.expr()
-y w_x(x, y) + x w_y(x, y)
```

The Euclidean norm of the vector field \( v \) is a scalar field on \( \mathbb{E}^2 \):

```python
sage: s = norm(v)
sage: s.display()
\( |v|: \mathbb{E}^2 \rightarrow \mathbb{R} \)
\( (x, y) \mapsto sqrt(x^2 + y^2) \)
```

Again, the corresponding symbolic expression is obtained via `expr()`:

```python
sage: s.expr()
sqrt(x^2 + y^2)
```

```python
sage: norm(w).expr()
sqrt(w_x(x, y)^2 + w_y(x, y)^2)
```
We have of course \[ ||v||^2 = v \cdot v \]

```
sage: norm(v)^2 == v.dot(v)
True
```

**Values at a given point**

We introduce a point \( p \in \mathbb{E}^2 \) via the generic SageMath syntax for creating an element from its parent (here \( \mathbb{E}^2 \)), i.e. the call operator \( () \), with the Cartesian coordinates of the point as the first argument:

```
sage: p = E((-2,3), name='p')
sage: p
Point p on the Euclidean plane \mathbb{E}^2
```

The coordinates of \( p \) are returned by the method `coord()`:

```
sage: p.coord()
(-2, 3)
```

or by letting the chart `cartesian` act on the point:

```
sage: cartesian(p)
(-2, 3)
```

The value of the scalar field \( s = \text{norm}(v) \) at \( p \) is:

```
sage: s(p)
sqrt(13)
```

The value of a vector field at \( p \) is obtained by the method `at()` (since the call operator \( () \) is reserved for the action on scalar fields, see `Vector fields as derivations` below):

```
sage: vp = v.at(p)
sage: vp
Vector v at Point p on the Euclidean plane \mathbb{E}^2
sage: vp.display()
v = -3 e_x - 2 e_y
sage: wp = w.at(p)
sage: wp.display()
w = w_x(-2, 3) e_x + w_y(-2, 3) e_y
sage: s = v.at(p) + pi*w.at(p)
sage: s.display()
(pi*w_x(-2, 3) - 3) e_x + (pi*w_y(-2, 3) - 2) e_y
```
3. Differential operators

The standard operators \( \text{grad} \), \( \text{div} \), etc. involved in vector calculus are accessible as methods on scalar fields and vector fields (e.g. \( \text{v}.\text{div}() \)). However, to use standard mathematical notations (e.g. \( \text{div}(v) \)), let us import the functions \( \text{grad}(), \text{div}() \), and \( \text{laplacian}() \) in the global namespace:

```
sage: from sage.manifolds.operators import *
```

### Divergence

The divergence of a vector field is returned by the function \( \text{div}() \); the output is a scalar field on \( \mathbb{R}^2 \):

```
sage: div(v)
Scalar field \( \text{div}(v) \) on the Euclidean plane \( \mathbb{R}^2 \)
sage: div(v).display()
div(v): \( \mathbb{R}^2 \to \mathbb{R} \)
\( (x, y) \mapsto 0 \)
```

In the present case, \( \text{div} \, v \) vanishes identically:

```
sage: div(v) == 0
True
```

On the contrary, the divergence of \( w \) is:

```
sage: div(w).display()
div(w): \( \mathbb{R}^2 \to \mathbb{R} \)
\( (x, y) \mapsto d(w_x)/dx + d(w_y)/dy \)
sage: div(w).expr()
diff(w_x(x, y), x) + diff(w_y(x, y), y)
```

### Gradient

The gradient of a scalar field, e.g. \( s = \text{norm}(v) \), is returned by the function \( \text{grad}() \); the output is a vector field:

```
sage: s = norm(v)
sage: grad(s)
Vector field \( \text{grad}(|v|) \) on the Euclidean plane \( \mathbb{R}^2 \)
sage: grad(s).display()
\( \text{grad}(|v|) = x/\sqrt{x^2 + y^2} \, e_x + y/\sqrt{x^2 + y^2} \, e_y \)
sage: grad(s)[2]
y/\sqrt{x^2 + y^2}
```

For a generic scalar field, like:

```
sage: F = E.scalar_field(function('f')(x,y), name='F')
```

we have:

```
sage: grad(F).display()
grad(F) = d(f)/dx \, e_x + d(f)/dy \, e_y
```

(continues on next page)
sage: grad(F)[:]
[d(f)/dx, d(f)/dy]

Of course, we may combine grad() and div():

sage: grad(div(w)).display()
grad(div(w)) = (d^2(w_x)/dx^2 + d^2(w_y)/dxdy) e_x + (d^2(w_x)/dxdy + d^2(w_y)/dy^2) e_y

**Laplace operator**

The Laplace operator \( \Delta \) is obtained by the function laplacian(); it acts on scalar fields:

sage: laplacian(F).display()
Delta(F): E^2 \rightarrow \mathbb{R}
\( (x, y) \mapsto d^2(f)/dx^2 + d^2(f)/dy^2 \)

as well as on vector fields:

sage: laplacian(w).display()
Delta(w) = (d^2(w_x)/dx^2 + d^2(w_x)/dy^2) e_x + (d^2(w_y)/dx^2 + d^2(w_y)/dy^2) e_y

For a scalar field, we have the identity

\[ \Delta F = \text{div} \left( \text{grad} \ F \right), \]

as we can check:

sage: laplacian(F) == div(grad(F))
True

### 4. Polar coordinates

Polar coordinates \((r, \phi)\) are introduced on \(E^2\) by:

sage: polar.<r,ph> = E.polar_coordinates()
sage: polar
Chart (E^2, (r, ph))
sage: polar.coord_range()
r: (0, +oo); ph: [0, 2*pi] (periodic)

They are related to Cartesian coordinates by the following transformations:

sage: E.coord_change(polar, cartesian).display()
x = r*cos(ph)
y = r*sin(ph)
sage: E.coord_change(cartesian, polar).display()
r = sqrt(x^2 + y^2)
ph = arctan2(y, x)

The orthonormal vector frame \((e_r, e_\phi)\) associated with polar coordinates is returned by the method polar_frame():
sage: polar_frame = E.polar_frame()
sage: polar_frame
Vector frame (E^2, (e_r,e_ph))

sage: er = polar_frame[1]
sage: er.display()
e_r = x/sqrt(x^2 + y^2) e_x + y/sqrt(x^2 + y^2) e_y

The above display is in the default frame (Cartesian frame) with the default coordinates (Cartesian). Let us ask for the
display in the same frame, but with the components expressed in polar coordinates:

sage: er.display(cartesian.frame(), polar)
e_r = cos(ph) e_x + sin(ph) e_y

Similarly:

sage: eph = polar_frame[2]
sage: eph.display()
e_ph = -y/sqrt(x^2 + y^2) e_x + x/sqrt(x^2 + y^2) e_y
sage: eph.display(cartesian.frame(), polar)
e_ph = -sin(ph) e_x + cos(ph) e_y

We may check that \((e_r, e_\phi)\) is an orthonormal frame:

sage: all([er.dot(er) == 1, er.dot(eph) == 0, eph.dot(eph) == 1])
True

Scalar fields can be expressed in terms of polar coordinates:

sage: F.display()
F: E^2 \rightarrow \mathbb{R}
(x, y) \mapsto f(x, y)
(r, ph) \mapsto f(r*cos(ph), r*sin(ph))
sage: F.display(polar)
F: E^2 \rightarrow \mathbb{R}
(r, ph) \mapsto f(r*cos(ph), r*sin(ph))

and we may ask for the components of vector fields in terms of the polar frame:

sage: v.display()  # default frame and default coordinates (both Cartesian ones)
v = -y e_x + x e_y
sage: v.display(polar_frame)  # polar frame and default coordinates
v = sqrt(x^2 + y^2) e_ph
sage: v.display(polar_frame, polar)  # polar frame and polar coordinates
v = r e_ph

sage: w.display()
w = w_x(x, y) e_x + w_y(x, y) e_y
sage: w.display(polar_frame, polar)
w = (cos(ph)*w_x(r*cos(ph), r*sin(ph)) + sin(ph)*w_y(r*cos(ph), r*sin(ph))) e_r
+ (-sin(ph)*w_x(r*cos(ph), r*sin(ph)) + cos(ph)*w_y(r*cos(ph), r*sin(ph))) e_ph
Gradient in polar coordinates

Let us define a generic scalar field in terms of polar coordinates:

```python
sage: H = E.scalar_field({'polar': function('h')(r, ph)}, name='H')
sage: H.display(polar)
H: E^2 \rightarrow \mathbb{R}
(r, ph) \mapsto h(r, ph)
```

The gradient of $H$ is then:

```python
sage: grad(H).display(polar_frame, polar)
grad(H) = d(h)/dr e_r + d(h)/dph/r e_ph
```

The access to individual components is achieved via the square bracket operator, where, in addition to the index, one has to specify the vector frame and the coordinates if they are not the default ones:

```python
sage: grad(H).display(cartesian.frame(), polar)
grad(H) = (r*cos(ph)*d(h)/dr - sin(ph)*d(h)/dph)/r e_x + (r*sin(ph)*d(h)/dr + cos(ph)*d(h)/dph)/r e_y
sage: grad(H)[polar_frame, 2, polar]
d(h)/dph/r
```

Divergence in polar coordinates

Let us define a generic vector field in terms of polar coordinates:

```python
sage: u = E.vector_field(function('u_r')(r, ph),
....: function('u_ph', latex_name=r'u_\phi')(r, ph),
....: frame=polar_frame, chart=polar, name='u')
sage: u.display(polar_frame, polar)
u = u_r(r, ph) e_r + u_ph(r, ph) e_ph
```

Its divergence is:

```python
sage: div(u).display(polar)
div(u): E^2 \rightarrow \mathbb{R}
(r, ph) \mapsto (r*d(u_r)/dr + u_r(r, ph) + d(u_ph)/dph)/r
sage: div(u).expr(polar)
div(u)(r, ph) = (r*diff(u_r(r, ph), r) + u_r(r, ph) + diff(u_ph(r, ph), ph))/r
sage: div(u).expr(polar).expand()
```

u_r(r, ph)/r + diff(u_ph(r, ph), ph)/r + diff(u_r(r, ph), r)
Using polar coordinates by default:

In order to avoid specifying the arguments `polar_frame` and `polar` in `display()`, `expr()` and [], we may change the default values by means of `set_default_chart()` and `set_default_frame()`:

```
sage: E.set_default_chart(polar)
sage: E.set_default_frame(polar_frame)
```

Then we have:

```
sage: u.display()
u = u_r(r, ph) e_r + u_ph(r, ph) e_ph
sage: u[1]
u_r(r, ph)
```

```
sage: v.display()
v = r e_ph
sage: v[2]
r
```

```
sage: w.display()
w = (cos(ph)*w_x(r*cos(ph), r*sin(ph)) + sin(ph)*w_y(r*cos(ph), r*sin(ph))) e_r + (- sin(ph)*w_x(r*cos(ph), r*sin(ph)) + cos(ph)*w_y(r*cos(ph), r*sin(ph))) e_ph
sage: div(u).expr()
(r*diff(u_r(r, ph), r) + u_r(r, ph) + diff(u_ph(r, ph), ph))/r
```

5. Advanced topics: the Euclidean plane as a Riemannian manifold

\( \mathbb{E}^2 \) is actually a Riemannian manifold (see pseudo_riemannian), i.e. a smooth real manifold endowed with a positive definite metric tensor:

```
sage: E.category()
Join of
    Category of smooth manifolds over Real Field with 53 bits of precision and
    Category of connected manifolds over Real Field with 53 bits of precision and
    Category of complete metric spaces
sage: E.base_field() is RR
True
```

Actually RR is used here as a proxy for the real field (this should be replaced in the future, see the discussion at trac ticket #24456) and the 53 bits of precision play of course no role for the symbolic computations.

The user atlas of \( \mathbb{E}^2 \) has two charts:

```
sage: E.atlas()
[Chart (\( \mathbb{E}^2 \), (x, y)), Chart (\( \mathbb{E}^2 \), (r, ph))]
```

while there are three vector frames defined on \( \mathbb{E}^2 \):

```
sage: E.frames()
[Coordinate frame (\( \mathbb{E}^2 \), (e_x,e_y)),
 Coordinate frame (\( \mathbb{E}^2 \), (\partial/\partial r,\partial/\partial ph)),
 Vector frame (\( \mathbb{E}^2 \), (e_r,e_ph))]
```
Indeed, there are two frames associated with polar coordinates: the coordinate frame \((\frac{\partial}{\partial r}, \frac{\partial}{\partial \phi})\) and the orthonormal frame \((e_r, e_\phi)\).

**Riemannian metric**

The default metric tensor of \(E^2\) is:

```python
sage: g = E.metric()
sage: g
Riemannian metric g on the Euclidean plane E^2
sage: g.display()
g = e^r \otimes e^r + e^\phi \otimes e^\phi
```

In the above display, \(e^r = e_r\) and \(e^\phi = e_\phi\) are the 1-forms defining the coframe dual to the orthonormal polar frame \((e_r, e_\phi)\), which is the default vector frame on \(E^2\):

```python
sage: polar_frame.coframe()
Coframe \((E^2, (e^r, e^\phi))\)
```

Of course, we may ask for some display with respect to frames different from the default one:

```python
sage: g.display(cartesian.frame())
g = dx \otimes dx + dy \otimes dy
sage: g.display(polar.frame())
g = dr \otimes dr + r^2 d\phi \otimes d\phi
sage: g[:]
[1 0]
[0 1]
sage: g[polar.frame(),:]
[ 1  0]
[ 0 r^2]
```

\(g\) is a flat metric: its (Riemann) curvature tensor (see \(riemann()\)) is zero:

```python
sage: g.riemann()
Tensor field Riem\(g\) of type (1,3) on the Euclidean plane E^2
sage: g.riemann().display()
Riem\(g\) = 0
```

The metric \(g\) is defining the dot product on \(E^2\):

```python
sage: v.dot(w) == g(v,w)
True
sage: norm(v) == sqrt(g(v,v))
True
```
Vector fields as derivations

Vector fields act as derivations on scalar fields:

```sage
v(F)
Scalar field v(F) on the Euclidean plane E^2
dF = F.differential()
dF
1-form dF on the Euclidean plane E^2
v(F) == dF(v)
True
```

The set $X(E^2)$ of all vector fields on $E^2$ is a free module of rank 2 over the commutative algebra of smooth scalar fields on $E^2$, $\mathcal{C}^\infty(E^2)$:

```sage
XE = v.parent()
XE
Free module X(E^2) of vector fields on the Euclidean plane E^2
XE.category()
Category of finite dimensional modules over Algebra of differentiable scalar fields on the Euclidean plane E^2
XE.base_ring()
Algebra of differentiable scalar fields on the Euclidean plane E^2
CE = F.parent()
CE
Algebra of differentiable scalar fields on the Euclidean plane E^2
CE is XE.base_ring()
True
CE.category()
Join of Category of commutative algebras over Symbolic Ring and Category of homsets of_
---topological spaces
rank(XE)
2
```

The bases of the free module $X(E^2)$ are nothing but the vector frames defined on $E^2$:

```sage
XE.bases()
[Coordinate frame (E^2, (e_x,e_y)),
 Coordinate frame (E^2, (\partial/\partial r, \partial/\partial ph)),
 Vector frame (E^2, (e_r,e_ph))]
```
Tangent spaces

A vector field evaluated at a point $p$ is a vector in the tangent space $T_p\mathbb{E}^2$:

```python
sage: vp = v.at(p)
sage: vp.display()
v = -3\, e_x - 2\, e_y
```

```python
sage: Tp = vp.parent()
sage: Tp
Tangent space at Point p on the Euclidean plane $\mathbb{E}^2$
sage: Tp.category()
Category of finite dimensional vector spaces over Symbolic Ring
sage: Tp.dimension()
2
sage: isinstance(Tp, FiniteRankFreeModule)
True
sage: sorted(Tp.bases(), key=str)
[Basis (e_r,e_ph) on the Tangent space at Point p on the Euclidean plane $\mathbb{E}^2$,
  Basis (e_x,e_y) on the Tangent space at Point p on the Euclidean plane $\mathbb{E}^2$]
```

Levi-Civita connection

The Levi-Civita connection associated to the Euclidean metric $g$ is:

```python
sage: nabla = g.connection()
sage: nabla
Levi-Civita connection nabla_g associated with the Riemannian metric g on the Euclidean plane $\mathbb{E}^2$
```

The corresponding Christoffel symbols with respect to the polar coordinates are:

```python
sage: g.christoffel_symbols_display()
\Gamma^r_{\phi \phi} = -r
\Gamma^\phi_{r \phi} = 1/r
```

By default, only nonzero and nonredundant values are displayed (for instance $\Gamma^\phi_{\phi r}$ is skipped, since it can be deduced from $\Gamma^\phi_{r \phi}$ by symmetry on the last two indices).

The Christoffel symbols with respect to the Cartesian coordinates are all zero:

```python
sage: g.christoffel_symbols_display(chart=cartesian, only_nonzero=False)
\Gamma^x_{xx} = 0
\Gamma^x_{xy} = 0
\Gamma^x_{yy} = 0
\Gamma^y_{xx} = 0
\Gamma^y_{xy} = 0
\Gamma^y_{yy} = 0
```

$\nabla_g$ is the connection involved in differential operators:
12.1.6 Linear Programming (Mixed Integer)

This document explains the use of linear programming (LP) – and of mixed integer linear programming (MILP) – in Sage by illustrating it with several problems it can solve. Most of the examples given are motivated by graph-theoretic concerns, and should be understandable without any specific knowledge of this field. As a tool in Combinatorics, using linear programming amounts to understanding how to reformulate an optimization (or existence) problem through linear constraints.

This is a translation of a chapter from the book Calcul mathematique avec Sage. This book now exists in English, too.

Definition

Here we present the usual definition of what a linear program is: it is defined by a matrix $A : \mathbb{R}^m \rightarrow \mathbb{R}^n$, along with two vectors $b, c \in \mathbb{R}^n$. Solving a linear program is searching for a vector $x$ maximizing an objective function and satisfying a set of constraints, i.e.

$$
c^t x = \max_{x' \text{ such that } Ax' \leq b} c^t x'
$$

where the ordering $u \leq u'$ between two vectors means that the entries of $u'$ are pairwise greater than the entries of $u$. We also write:

Max: $c^t x$

Such that: $Ax \leq b$

Equivalently, we can also say that solving a linear program amounts to maximizing a linear function defined over a polytope (preimage or $A^{-1}(\leq b)$). These definitions, however, do not tell us how to use linear programming in combinatorics. In the following, we will show how to solve optimization problems like the Knapsack problem, the Maximum Matching problem, and a Flow problem.

Mixed integer linear programming

There are bad news coming along with this definition of linear programming: an LP can be solved in polynomial time. This is indeed bad news, because this would mean that unless we define LP of exponential size, we cannot expect LP to solve NP-complete problems, which would be a disappointment. On a brighter side, it becomes NP-complete to solve a linear program if we are allowed to specify constraints of a different kind: requiring that some variables be integers instead of real values. Such an LP is actually called a “mixed integer linear program” (some variables can be integers, some other reals). Hence, we can expect to find in the MILP framework a wide range of expressivity.
Practical

The MILP class

The MILP class in Sage represents a MILP! It is also used to solve regular LP. It has a very small number of methods, meant to define our set of constraints and variables, then to read the solution found by the solvers once computed. It is also possible to export a MILP defined with Sage to a .lp or .mps file, understood by most solvers.

Let us ask Sage to solve the following LP:

Max: \( x + y + 3z \)
Such that:
\[
\begin{align*}
x + 2y &\leq 4 \\
5z - y &\leq 8 \\
x, y, z &\geq 0
\end{align*}
\]

To achieve it, we need to define a corresponding MILP object, along with 3 variables \(x\), \(y\) and \(z\):

```
sage: p = MixedIntegerLinearProgram()
sage: v = p.new_variable(real=True, nonnegative=True)
sage: x, y, z = v['x'], v['y'], v['z']
```

Next, we set the objective function

```
sage: p.set_objective(x + y + 3*z)
```

And finally we set the constraints

```
sage: p.add_constraint(x + 2*y <= 4)
sage: p.add_constraint(5*z - y <= 8)
```

The `solve` method returns by default the optimal value reached by the objective function

```
sage: round(p.solve(), 2)
8.8
```

We can read the optimal assignment found by the solver for \(x\), \(y\) and \(z\) through the `get_values` method

```
sage: round(p.get_values(x), 2)
4.0
sage: round(p.get_values(y), 2)
0.0
sage: round(p.get_values(z), 2)
1.6
```

Variables

In the previous example, we obtained variables through \(v['x'], v['y']\) and \(v['z']\). This being said, larger LP/MILP will require us to associate an LP variable to many Sage objects, which can be integers, strings, or even the vertices and edges of a graph. For example:

```
sage: x = p.new_variable(real=True, nonnegative=True)
```

With this new object \(x\) we can now write constraints using \(x[1], \ldots, x[15]\).
Notice that we did not need to define the “length” of \( x \). Actually, \( x \) would accept any immutable object as a key, as a dictionary would. We can now write
\[
\text{sage: } p.add_constraint(x[\text{"I am a valid key"}] + x[\text{"a",\pi}] \leq 3)
\]
And because any immutable object can be used as a key, doubly indexed variables \( x_{1,1}, \ldots, x_{1,15}, x_{2,1}, \ldots, x_{15,15} \) can be referenced by \( x[1,1], \ldots, x[1,15], x[2,1], \ldots, x[15,15] \)

\[
\text{sage: } p.add_constraint(x[3,2] + x[5] == 6)
\]

### Typed variables and bounds

**Types:** If you want a variable to assume only integer or binary values, use the `integer=True` or `binary=True` arguments of the `new_variable` method. Alternatively, call the `set_integer` and `set_binary` methods.

**Bounds:** If you want your variables to only take nonnegative values, you can say so when calling `new_variable` with the argument `nonnegative=True`. If you want to set a different upper/lower bound on a variable, add a constraint or use the `set_min`, `set_max` methods.

### Basic linear programs

#### Knapsack

The Knapsack problem is the following: given a collection of items having both a weight and a usefulness, we would like to fill a bag whose capacity is constrained while maximizing the usefulness of the items contained in the bag (we will consider the sum of the items’ usefulness). For the purpose of this tutorial, we set the restriction that the bag can only carry a certain total weight.

To achieve this, we have to associate to each object \( o \) of our collection \( C \) a binary variable \( \text{taken}[o] \), set to 1 when the object is in the bag, and to 0 otherwise. We are trying to solve the following MILP

\[
\begin{align*}
\text{Max: } & \sum_{o \in L} \text{usefulness}_o \times \text{taken}_o \\
\text{Such that: } & \sum_{o \in L} \text{weight}_o \times \text{taken}_o \leq C
\end{align*}
\]

Using Sage, we will give to our items a random weight:

\[
\text{sage: } C = 1
\]
\[
\text{sage: } L = [\text{"pan", "book", "knife", "gourd", "flashlight"]}
\]
\[
\text{sage: } L.extend([\text{"random_stuff_" + str(i) for i in range(20)}])
\]
\[
\text{sage: weight = {}}
\]
\[
\text{sage: usefulness = {}}
\]
We can now define the MILP itself

```python
sage: p = MixedIntegerLinearProgram()
```

```python
sage: taken = p.new_variable(binary=True)
```

```python
sage: p.add_constraint(p.sum(weight[o] * taken[o] for o in L) <= C)
```

```python
sage: p.set_objective(p.sum(usefulness[o] * taken[o] for o in L))
```

```python
sage: p.solve()  # abs tol 1e-6
3.1502766806530307
```

```python
sage: taken = p.get_values(taken)
```

The solution found is (of course) admissible

```python
sage: sum(weight[o] * taken[o] for o in L)  # abs tol 1e-6
0.6964959796619171
```

Should we take a flashlight?

```python
sage: taken["flashlight"]  # abs tol 1e-6
1.0
```

Wise advice. Based on purely random considerations.

**Matching**

Given a graph $G$, a matching is a set of pairwise disjoint edges. The empty set is a trivial matching. So we focus our attention on maximum matchings: we want to find in a graph a matching whose cardinality is maximal. Computing the maximum matching in a graph is a polynomial problem, which is a famous result of Edmonds. Edmonds’ algorithm is based on local improvements and the proof that a given matching is maximum if it cannot be improved. This algorithm is not the hardest to implement among those graph theory can offer, though this problem can be modeled with a very simple MILP.

To do it, we need – as previously – to associate a binary variable to each one of our objects: the edges of our graph (a value of 1 meaning that the corresponding edge is included in the maximum matching). Our constraint on the edges taken being that they are disjoint, it is enough to require that, $x$ and $y$ being two edges and $m_x, m_y$ their associated variables, the inequality $m_x + m_y \leq 1$ is satisfied, as we are sure that the two of them cannot both belong to the matching. Hence, we are able to write the MILP we want. However, the number of inequalities can be easily decreased by noticing that two edges cannot be taken simultaneously inside a matching if and only if they have a common endpoint $v$. We can then require instead that at most one edge incident to $v$ be taken inside the matching, which is a linear constraint. We will be solving:

$$\text{Max: } \sum_{e \in E(G)} m_e$$

$$\text{Such that: } \forall v, \sum_{e \in E(G), v \sim e} m_e \leq 1$$
Let us write the Sage code of this MILP:

```sage
sage: g = graphs.PetersenGraph()
sage: p = MixedIntegerLinearProgram()
sage: matching = p.new_variable(binary=True)

sage: p.set_objective(p.sum(matching[e] for e in g.edges(sort=False, labels=False)))

sage: for v in g:
    ....:     p.add_constraint(p.sum(matching[e] for e in g.edges_incident(v, labels=False)) <= 1)

sage: p.solve()
5.0

sage: matching = p.get_values(matching, convert=bool, tolerance=1e-3)
sage: sorted(e for e, b in matching.items() if b)  # random
[(0, 4), (1, 6), (2, 3), (5, 8), (7, 9)]
sage: len(_)
5
```

**Flows**

Yet another fundamental algorithm in graph theory: maximum flow! It consists, given a directed graph and two vertices $s, t$, in sending a maximum flow from $s$ to $t$ using the edges of $G$, each of them having a maximal capacity.
The definition of this problem is almost its LP formulation. We are looking for real values associated to each edge, which would represent the intensity of flow going through them, under two types of constraints:

- The amount of flow arriving on a vertex (different from \(s\) or \(t\)) is equal to the amount of flow leaving it.
- The amount of flow going through an edge is bounded by the capacity of this edge.

This being said, we have to maximize the amount of flow leaving \(s\): all of it will end up in \(t\), as the other vertices are sending just as much as they receive. We can model the flow problem with the following LP

Max: \(\sum_{sv \in G} f_{sv}\)

Such that: \(\forall v \in G, v \neq s, t\) \(\sum_{vu \in G} f_{vu} - \sum_{uv \in G} f_{uv} = 0\)

\(\forall uv \in G, f_{uv} \leq 1\)

We will solve the flow problem on an orientation of Chvatal’s graph, in which all the edges have a capacity of 1:

```python
sage: g = graphs.ChvatalGraph()
sage: g = g.minimum_outdegree_orientation()

sage: p = MixedIntegerLinearProgram()
sage: f = p.new_variable(real=True, nonnegative=True)

sage: s, t = 0, 2
```
sage: for v in g:
....:     if v != s and v != t:
....:         p.add_constraint(
....:             p.sum(f[v,u] for u in g.neighbors_out(v))
....:             - p.sum(f[u,v] for u in g.neighbors_in(v)) == 0)

sage: for e in g.edges(sort=False, labels=False):
....:     p.add_constraint(f[e] <= 1)

sage: p.set_objective(p.sum(f[s,u] for u in g.neighbors_out(s)))

sage: p.solve()  # rel tol 2e-11
2.0
Sage solves linear programs by calling specific libraries. The following libraries are currently supported:

- **CBC**: A solver from COIN-OR, provided under the Eclipse Public License (EPL), which is an open source license but incompatible with GPL. CBC and the Sage CBC backend can be installed using the shell command:

  ```
  $ sage -i -c sage_numerical_backends_coin
  ```

- **CPLEX**: Proprietary, but available for free for researchers and students through IBM’s Academic Initiative. Since trac ticket #27790, only versions 12.8 and above are supported.

  Install CPLEX according to the instructions on the website, which includes obtaining a license key.

  Then find the installation directory of your ILOG CPLEX Studio installation, which contains subdirectories `cplex`, `doc`, `opl`, etc. Set the environment variable `CPLEX_HOME` to this directory; for example using the following shell command (on macOS):

  ```
  $ export CPLEX_HOME=/Applications/CPLEX_Studio1210
  ```

  or (on Linux):

  ```
  $ export CPLEX_HOME=/opt/ibm/ILOG/CPLEX_Studio1210
  ```

  Now verify that the CPLEX binary that you will find in the subdirectory `cplex/bin/ARCH-OS` starts correctly, for example:

  ```
  $ export CPLEX_HOME=/Applications/CPLEX_Studio1210
  $ CPLEX_HOME/cplex/bin/x86-64_osx/cplex
  Welcome to IBM(R) ILOG(R) CPLEX(R) Interactive Optimizer...
  ```

  This environment variable only needs to be set for the following step: Install the Sage CPLEX backend using the shell command:

  ```
  $ sage -i -c sage_numerical_backends_cplex
  ```

- **CVXOPT**: an LP solver from Python Software for Convex Optimization, uses an interior-point method, always installed in Sage.

- **GLPK**: A solver from GNU

  Licensed under the GPLv3. This solver is always installed, as the default one, in Sage.

- **Gurobi**: Proprietary, but available for free for researchers and students via Gurobi’s Academic Program.

  Install Gurobi according to the instructions on the website, which includes obtaining a license key. The installation should make the interactive Gurobi shell `gurobi.sh` available in your PATH. Verify this by typing the shell command `gurobi.sh`:

  ```
  $ gurobi.sh
  ...
  Gurobi Interactive Shell (mac64), Version 9.0.0
  Copyright (c) 2019, Gurobi Optimization, LLC
  Type "help()" for help
  gurobi>
  ```
If this does not work, adjust your PATH or create symbolic links so that `gurobi.sh` is found.

Now install the Sage Gurobi backend using the shell command:

```
$ sage -i -c sage_numerical_backends_gurobi
```

- **PPL**: A solver from bugSeng.

  This solver provides exact (arbitrary precision) computation, always installed in Sage.

  Licensed under the GPLv3.

### 12.1.7 Number Theory and the RSA Public Key Cryptosystem

**Author: Minh Van Nguyen <nguyenminh2@gmail.com>**

This tutorial uses Sage to study elementary number theory and the RSA public key cryptosystem. A number of Sage commands will be presented that help us to perform basic number theoretic operations such as greatest common divisor and Euler’s phi function. We then present the RSA cryptosystem and use Sage’s built-in commands to encrypt and decrypt data via the RSA algorithm. Note that this tutorial on RSA is for pedagogy purposes only. For further details on cryptography or the security of various cryptosystems, consult specialized texts such as [MenezesEtAl1996], [Stinson2006], and [TrappeWashington2006].

**Elementary number theory**

We first review basic concepts from elementary number theory, including the notion of primes, greatest common divisors, congruences and Euler’s phi function. The number theoretic concepts and Sage commands introduced will be referred to in later sections when we present the RSA algorithm.

**Prime numbers**

Public key cryptography uses many fundamental concepts from number theory, such as prime numbers and greatest common divisors. A positive integer \( n > 1 \) is said to be prime if its factors are exclusively 1 and itself. In Sage, we can obtain the first 20 prime numbers using the command `primes_first_n`:

```
sage: primes_first_n(20)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71]
```

**Greatest common divisors**

Let \( a \) and \( b \) be integers, not both zero. Then the greatest common divisor (GCD) of \( a \) and \( b \) is the largest positive integer which is a factor of both \( a \) and \( b \). We use \( \gcd(a, b) \) to denote this largest positive factor. One can extend this definition by setting \( \gcd(0, 0) = 0 \). Sage uses \( \gcd(a, b) \) to denote the GCD of \( a \) and \( b \). The GCD of any two distinct primes is 1, and the GCD of 18 and 27 is 9.

```
sage: gcd(3, 59)
1
sage: gcd(18, 27)
9
```

If \( \gcd(a, b) = 1 \), we say that \( a \) is coprime (or relatively prime) to \( b \). In particular, \( \gcd(3, 59) = 1 \) so 3 is coprime to 59 and vice versa.
Congruences

When one integer is divided by a non-zero integer, we usually get a remainder. For example, upon dividing 23 by 5, we get a remainder of 3; when 8 is divided by 5, the remainder is again 3. The notion of congruence helps us to describe the situation in which two integers have the same remainder upon division by a non-zero integer. Let \( a, b, n \in \mathbb{Z} \) such that \( n \neq 0 \). If \( a \) and \( b \) have the same remainder upon division by \( n \), then we say that \( a \) is congruent to \( b \) modulo \( n \) and denote this relationship by

\[
a \equiv b \pmod{n}
\]

This definition is equivalent to saying that \( n \) divides the difference of \( a \) and \( b \), i.e. \( n \mid (a - b) \). Thus \( 23 \equiv 8 \pmod{5} \) because when both 23 and 8 are divided by 5, we end up with a remainder of 3. The command \texttt{mod} allows us to compute such a remainder:

\begin{verbatim}
sage: mod(23, 5)
3
sage: mod(8, 5)
3
\end{verbatim}

Euler’s phi function

Consider all the integers from 1 to 20, inclusive. List all those integers that are coprime to 20. In other words, we want to find those integers \( n \), where \( 1 \leq n \leq 20 \), such that \( \gcd(n, 20) = 1 \). The latter task can be easily accomplished with a little bit of Sage programming:

\begin{verbatim}
sage: L = []
sage: for n in range(1, 21):
    ....:     if gcd(n, 20) == 1:
    ....:         L.append(n)

sage: L
[1, 3, 7, 9, 11, 13, 17, 19]
\end{verbatim}

The above programming statements can be saved to a text file called, say, \texttt{/home/mvngu/totient.sage}, organizing it as follows to enhance readability.

\begin{verbatim}
L = []
for n in range(1, 21):
    if gcd(n, 20) == 1:
        L.append(n)

L
[1, 3, 7, 9, 11, 13, 17, 19]
\end{verbatim}

We refer to \texttt{totient.sage} as a Sage script, just as one would refer to a file containing Python code as a Python script. We use 4 space indentations, which is a coding convention in Sage as well as Python programming, instead of tabs.

The command \texttt{load} can be used to read the file containing our programming statements into Sage and, upon loading the content of the file, have Sage execute those statements:

\begin{verbatim}
load("/home/mvngu/totient.sage")
[1, 3, 7, 9, 11, 13, 17, 19]
\end{verbatim}

From the latter list, there are 8 integers in the closed interval \([1, 20]\) that are coprime to 20. Without explicitly generating the list.
how can we compute the number of integers in \([1, 20]\) that are coprime to 20? This is where Euler’s phi function comes in handy. Let \(n \in \mathbb{Z}\) be positive. Then Euler’s phi function counts the number of integers \(a\), with \(1 \leq a \leq n\), such that \(\gcd(a, n) = 1\). This number is denoted by \(\varphi(n)\). Euler’s phi function is sometimes referred to as Euler’s totient function, hence the name \texttt{totient.sage} for the above Sage script. The command \texttt{euler_phi} implements Euler’s phi function. To compute \(\varphi(20)\) without explicitly generating the above list, we proceed as follows:

```
sage: euler_phi(20)
8
```

**How to keep a secret?**

Cryptography is the science (some might say art) of concealing data. Imagine that we are composing a confidential email to someone. Having written the email, we can send it in one of two ways. The first, and usually convenient, way is to simply press the send button and not care about how our email will be delivered. Sending an email in this manner is similar to writing our confidential message on a postcard and post it without enclosing our postcard inside an envelope. Anyone who can access our postcard can see our message. On the other hand, before sending our email, we can scramble the confidential message and then press the send button. Scrambling our message is similar to enclosing our postcard inside an envelope. While not 100% secure, at least we know that anyone wanting to read our postcard has to open the envelope.

In cryptography parlance, our message is called plaintext. The process of scrambling our message is referred to as encryption. After encrypting our message, the scrambled version is called ciphertext. From the ciphertext, we can recover our original unscrambled message via decryption. A cryptosystem is comprised of a pair of related encryption and decryption processes.

```
+ ---------+ encrypt +------------+ decrypt +-----------+
| plaintext| -----------> | ciphertext | -----------> | plaintext |
+----------+ +------------+ +-----------+
```

The following table provides a very simple method of scrambling a message written in English and using only upper case letters, excluding punctuation characters.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
<th>K</th>
<th>L</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>65</td>
<td>66</td>
<td>67</td>
<td>68</td>
<td>69</td>
<td>70</td>
<td>71</td>
<td>72</td>
<td>73</td>
<td>74</td>
<td>75</td>
<td>76</td>
<td>77</td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td>N</td>
<td>O</td>
<td>P</td>
<td>Q</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>U</td>
<td>V</td>
<td>W</td>
<td>X</td>
<td>Y</td>
<td>Z</td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td>78</td>
<td>79</td>
<td>80</td>
<td>81</td>
<td>82</td>
<td>83</td>
<td>84</td>
<td>85</td>
<td>86</td>
<td>87</td>
<td>88</td>
<td>89</td>
<td>90</td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
</tr>
</tbody>
</table>

Formally, let

\[
\Sigma = \{A, B, C, \ldots, Z\}
\]

be the set of capital letters of the English alphabet. Furthermore, let

\[
\Phi = \{65, 66, 67, \ldots, 90\}
\]

be the American Standard Code for Information Interchange (ASCII) encodings of the upper case English letters. Then the above table explicitly describes the mapping \(f : \Sigma \rightarrow \Phi\). (For those familiar with ASCII, \(f\) is actually a common process for encoding elements of \(\Sigma\), rather than a cryptographic “scrambling” process per se.) To scramble a message written using the alphabet \(\Sigma\), we simply replace each capital letter of the message with its corresponding
Keeping a secret with two keys

The Rivest, Shamir, Adleman (RSA) cryptosystem is an example of a public key cryptosystem. RSA uses a public key to encrypt messages and decryption is performed using a corresponding private key. We can distribute our public keys, but for security reasons we should keep our private keys to ourselves. The encryption and decryption processes draw upon techniques from elementary number theory. The algorithm below is adapted from page 165 of [TrappeWashington2006]. It outlines the RSA procedure for encryption and decryption.

1. Choose two primes $p$ and $q$ and let $n = pq$.
2. Let $e \in \mathbb{Z}$ be positive such that $\gcd(e, \varphi(n)) = 1$.
3. Compute a value for $d \in \mathbb{Z}$ such that $de \equiv 1 \pmod{\varphi(n)}$.
4. Our public key is the pair $(n, e)$ and our private key is the triple $(p, q, d)$.
5. For any non-zero integer $m < n$, encrypt $m$ using $c \equiv m^e \pmod{n}$.
6. Decrypt $c$ using $m \equiv c^d \pmod{n}$.

The next two sections will step through the RSA algorithm, using Sage to generate public and private keys, and perform encryption and decryption based on those keys.

Generating public and private keys

Positive integers of the form $M_m = 2^m - 1$ are called Mersenne numbers. If $p$ is prime and $M_p = 2^p - 1$ is also prime, then $M_p$ is called a Mersenne prime. For example, 31 is prime and $M_31 = 2^{31} - 1$ is a Mersenne prime, as can be verified using the command `is_prime(p)`. This command returns True if its argument $p$ is precisely a prime number; otherwise it returns False. By definition, a prime must be a positive integer, hence `is_prime(-2)` returns False although we know that 2 is prime. Indeed, the number $M_{61} = 2^{61} - 1$ is also a Mersenne prime. We can use $M_{31}$ and $M_{61}$ to work through step 1 in the RSA algorithm:

```sage
sage: p = (2^31) - 1
sage: is_prime(p)
True
sage: q = (2^61) - 1
sage: is_prime(q)
True
sage: n = p * q ; n
4951760154835678088235319297
```

A word of warning is in order here. In the above code example, the choice of $p$ and $q$ as Mersenne primes, and with so many digits far apart from each other, is a very bad choice in terms of cryptographic security. However, we shall use the above chosen numeric values for $p$ and $q$ for the remainder of this tutorial, always bearing in mind that they have been chosen for pedagogy purposes only. Refer to [MenezesEtAl1996], [Stinson2006], and [TrappeWashington2006] for in-depth discussions on the security of RSA, or consult other specialized texts.

For step 2, we need to find a positive integer that is coprime to $\varphi(n)$. The set of integers is implemented within the Sage module `sage.rings.integer_ring`. Various operations on integers can be accessed via the ZZ.* family of functions. For instance, the command `ZZ.random_element(n)` returns a pseudo-random integer uniformly distributed within the closed interval $[0, n - 1]$.

We can compute the value $\varphi(n)$ by calling the sage function `euler_phi(n)`, but for arbitrarily large prime numbers $p$ and $q$, this can take an enormous amount of time. Indeed, the private key can be quickly deduced from the public key.
once you know \( \varphi(n) \), so it is an important part of the security of the RSA cryptosystem that \( \varphi(n) \) cannot be computed in a short time, if only \( n \) is known. On the other hand, if the private key is available, we can compute \( \varphi(n) = (p-1)(q-1) \) in a very short time.

Using a simple programming loop, we can compute the required value of \( e \) as follows:

```plaintext
sage: p = (2^31) - 1
sage: q = (2^61) - 1
sage: n = p * q
sage: phi = (p - 1)*(q - 1); phi
4951760154835678088235319297
sage: e = ZZ.random_element(phi)
\...
\...
```

As \( e \) is a pseudo-random integer, its numeric value changes after each execution of \( e \) = ZZ.random_element(phi).

To calculate a value for \( d \) in step 3 of the RSA algorithm, we use the extended Euclidean algorithm. By definition of congruence, \( de \equiv 1 \pmod{\varphi(n)} \) is equivalent to

\[
de - k \cdot \varphi(n) = 1
\]

where \( k \in \mathbb{Z} \). From steps 1 and 2, we already know the numeric values of \( e \) and \( \varphi(n) \). The extended Euclidean algorithm allows us to compute \( d \) and \(-k\). In Sage, this can be accomplished via the command `xgcd`. Given two integers \( x \) and \( y \), `xgcd(x, y)` returns a 3-tuple \((g, s, t)\) that satisfies the Bézout identity \( g = \gcd(x, y) = sx + ty \). Having computed a value for \( d \), we then use the command `mod(d*e, phi)` to check that \( d*e \) is indeed congruent to 1 modulo \( \varphi(n) \).

```plaintext
sage: n = 4951760154835678088235319297
sage: e = 185056762330615966303954877
sage: phi = 4951760154835678088235319297
sage: bezout = xgcd(e, phi); bezout
\# random
(1, 4460824882019967172592779313, -1667095708515377925087033035)
sage: d = Integer(mod(bezout[1], phi)); d \# random
4460824882019967172592779313
sage: mod(d * e, phi)
1
```

Thus, our RSA public key is

\[
(n, e) = (4951760154835678088235319297, 185056762330615966303954877)
\]

and our corresponding private key is

\[
(p, q, d) = (2147483647, 2305843009213693951, 4460824882019967172592779313)
\]
Encryption and decryption

Suppose we want to scramble the message HELLOWORLD using RSA encryption. From the above ASCII table, our message maps to integers of the ASCII encodings as given below.

| H E L L O W O R L D |
| 72 69 76 76 79 87 79 82 76 68 |

Concatenating all the integers in the last table, our message can be represented by the integer

\[ m = 72697676798779827668 \]

There are other more cryptographically secure means for representing our message as an integer. The above process is used for demonstration purposes only and we strongly discourage its use in practice. In Sage, we can obtain an integer representation of our message as follows:

```
sage: m = "HELLOWORLD"
sage: m = [ord(x) for x in m]; m
[72, 69, 76, 76, 79, 87, 79, 82, 76, 68]
sage: m = ZZ(list(reversed(m)), 100) ; m
72697676798779827668
```

To encrypt our message, we raise \( m \) to the power of \( e \) and reduce the result modulo \( n \). The command \( \text{mod}(a^b, n) \) first computes \( a^b \) and then reduces the result modulo \( n \). If the exponent \( b \) is a “large” integer, say with more than 20 digits, then performing modular exponentiation in this naive manner takes quite some time. Brute force (or naive) modular exponentiation is inefficient and, when performed using a computer, can quickly consume a huge quantity of the computer’s memory or result in overflow messages. For instance, if we perform naive modular exponentiation using the command \( \text{mod}(m^e, n) \), where \( m, n \) and \( e \) are as given above, we would get an error message similar to the following:

```
mod(m^e, n)
Traceback (most recent call last)
...<ipython console> in <module>()
.../sage/rings/integer.so
in sage.rings.integer.Integer.__pow__ (sage/rings/integer.c:9650)()
RuntimeError: exponent must be at most 2147483647
```

There is a trick to efficiently perform modular exponentiation, called the method of repeated squaring, cf. page 879 of [CormenEtAl2001]. Suppose we want to compute \( a^b \mod n \). First, let \( d := 1 \) and obtain the binary representation of \( b \), say \( (b_1, b_2, \ldots, b_k) \) where each \( b_i \in \mathbb{Z}/2\mathbb{Z} \). For \( i := 1, \ldots, k \), let \( d := d^2 \mod n \) and if \( b_i = 1 \) then let \( d := da \mod n \). This algorithm is implemented in the function `power_mod`. We now use the function `power_mod` to encrypt our message:

```
sage: m = 72697676798779827668
sage: e = 1850567623300615966303954877
sage: n = 4951760154835678088235319297
sage: c = power_mod(m, e, n); c
630913632577520058415521090
```

Thus \( c = 630913632577520058415521090 \) is the ciphertext. To recover our plaintext, we raise \( c \) to the power of \( d \) and reduce the result modulo \( n \). Again, we use modular exponentiation via repeated squaring in the decryption process:
Notice in the last output that the value 72697676798779827668 is the same as the integer that represents our original message. Hence we have recovered our plaintext.

Acknowledgements

1. 2009-07-25: Ron Evans (Department of Mathematics, UCSD) reported a typo in the definition of greatest common divisors. The revised definition incorporates his suggestions.

2. 2008-11-04: Martin Albrecht (Information Security Group, Royal Holloway, University of London), John Cremona (Mathematics Institute, University of Warwick) and William Stein (Department of Mathematics, University of Washington) reviewed this tutorial. Many of their invaluable suggestions have been incorporated into this document.

Bibliography

12.1.8 Coding Theory in Sage


This tutorial, designed for beginners who want to discover how to use Sage for their work (research, experimentation, teaching) on coding theory, will present several key features of Sage’s coding theory library and explain how to find classes and methods you look for.

During this tutorial, we will cover the following parts:

- what you can do with generic linear codes and associated methods,
- what you can do with structured code families,
- what you can do to encode and recover messages, correct errors and
- what you can do to easily add errors to codewords,
- what you can do with codes in general.

The goal of this tutorial is to give a quick overview of what can be done with the library and how to use the main functionalities. It is neither a comprehensive description of all methods nor of specific classes. If one needs some specific information on the behaviour of a class/method related to coding theory, one should check the documentation for this class/method.
I. Generic Linear codes and associated methods

Let us start with the most basic code one can build: a generic linear code without any specific structure.

To build such a code, one just need to provide its generator matrix, as follows:

```python
sage: G = matrix(GF(3), [[1, 0, 0, 0, 1, 2, 1], 
....: [0, 1, 0, 0, 2, 1, 0], 
....: [0, 0, 1, 2, 2, 2, 2]])
sage: C = LinearCode(G)
```

With these lines, you just created a linear code, congratulations! Note that if you pass a matrix which is not full rank, Sage will turn it into a full-rank matrix before building the code, as illustrated in the following example:

```python
sage: G = matrix(GF(3), [[1, 0, 0, 0, 1, 2, 1], 
....: [0, 1, 0, 0, 2, 1, 0], 
....: [0, 0, 1, 2, 2, 2, 2], 
....: [1, 0, 1, 2, 0, 1, 0]]) #r3 = r0 + r2
sage: C = LinearCode(G)
sage: C.generator_matrix()
[1 0 0 1 2 1]
[0 1 0 2 1 0]
[0 0 1 2 2 2]
```

We now have a linear code… What can we do with it? As we can a lot of things, let us start with the basic functionalities.

In the example just above, we already asked for the code’s generator matrix. It is also possible to ask the code for its basic parameters: its `length` and `dimension` as illustrated thereafter:

```python
sage: C.length()
7
sage: C.dimension()
3
```

It is also possible to ask for our code’s minimum distance:
Of course, as C is a generic linear code, an exhaustive search algorithm is run to find the minimum distance, which will be slower and slower as the code grows.

By just typing the name of our code, we get a sentence which briefly describes it and gives its parameters:

```
sage: C
[7, 3] linear code over GF(3)
```

As the aim of this tutorial is not to give a comprehensive view of the methods, we won’t describe any other methods. If one wants to get all methods that can be run on a linear code, one can:

- either check the manual page of the file `Generic structures for linear codes over the Hamming metric`
- or type:

```
C.<tab>
```

in Sage to get a list of all available methods for C. Afterwards, typing:

```
C.method?
```

will show the manual page for method.

**Note:** Some generic methods require the installation of the optional package Guava for Gap. While some work is done to always propose a default implementation which *does not* require an optional package, there exist some methods which are not up to date - yet. If you’re receiving an error message related to Gap, please check the documentation of the method to verify if Guava has to be installed.

### II. Structured code families and an overview of the encoding and decoding system

#### II.1 Create specific codes in Sage

Now that we know how to create generic codes, we want to go deeper and create specific code families. In Sage, all codes families can be accessed by typing:

```
codes.<tab>
```

Doing so, you will get the comprehensive list of all code families Sage can build.

For the rest of this section, we will illustrate specific functionalities of these code families by manipulating `sage.coding.grs_code.GeneralizedReedSolomonCode`.

So, for starters, we want to create a Generalized Reed-Solomon (GRS) code.

By clicking on the link provided above, or typing:

```
codes.GeneralizedReedSolomonCode?
```

one can access the documentation page for GRS codes, find a definition of these and learn what is needed to build one in Sage.

Here we choose to build a [12, 6] GRS code over $\mathbb{F}_{13}$. To do this, we need up to three elements:
• The list of evaluation points,
• the dimension of the code, and
• optionally, the list of column multipliers.

We build our code as follows:

```
sage: F = GF(13)
sage: length, dimension = 12, 6
sage: evaluation_pts = F.list()[:length]
sage: column_mults = F.list()[1:length+1]
sage: C = codes.GeneralizedReedSolomonCode(evaluation_pts, dimension, column_mults)
```

Our GRS code is now created. We can ask for its parameters, as we did in the previous section:

```
sage: C.length()
12
sage: C.dimension()
6
sage: C.base_ring()
Finite Field of size 13
```

It is also possible to ask for the evaluation points and the column multipliers by calling `sage.coding.grs_code.GeneralizedReedSolomonCode.evaluation_points()` and `sage.coding.grs_code.GeneralizedReedSolomonCode.column_multipliers()`.

Now, if you know some theory for GRS codes, you know that it’s especially easy to compute their minimum distance, which is: \(d = n - k + 1\), where \(n\) is the length of the code and \(k\) is the dimension of the code.

Because Sage knows \(C\) is a GRS code, it will not run the exhaustive search algorithm presented in section I to find \(C\)'s minimum distance but use the operation introduced above. And you instantly get:

```
sage: C.minimum_distance()
7
```

All these parameters are summarized inside the string representation of our code:

```
sage: C
[12, 6, 7] Generalized Reed-Solomon Code over GF(13)
```

**Note:** Writing proper classes for code families is a work in progress. Some constructions under `codes.<tab>` might thus be functions which build a generic linear code, and in that case are only able to use generic algorithms. Please refer to the documentation of a construction to check if it is a function or a class.

### II.2 Encode and decode in Sage

In the previous part, we learnt how to find specific code families in Sage and create instances of these families.

In this part, we will learn how to encode and decode.

First of all, we want to generate a codeword to play with. There are two different ways to do that:

• It is possible to just generate a random element of our code, as follows:
Alternatively, we can create a message and then encode it into a codeword:

```
sage: msg = random_vector(C.base_field(), C.dimension())
sage: c = C.encode(msg)
sage: c in C
True
```

Either way, we obtained a codeword. So, we might want to put some errors in it, and try to correct these errors afterwards. We can obviously do it by changing the values at some random positions of our codeword, but we propose here something more general: communication channels. `sage.coding.channel.Channel` objects are meant as abstractions for communication channels and for manipulation of data representation. In this case, we want to emulate a communication channel which adds some, but not too many, errors to a transmitted word:

```
sage: err = 3
sage: Chan = channels.StaticErrorRateChannel(C.ambient_space(), err)
sage: Chan
Static error rate channel creating 3 errors, of input and output space Vector space of dimension 12 over Finite Field of size 13
sage: r = Chan.transmit(c)
sage: len((c-r).nonzero_positions())
3
```

If you want to learn more on Channels, please refer to section IV of this tutorial.

Thanks to our channel, we got a “received word”, `r`, as a codeword with errors on it. We can try to correct the errors and recover the original codeword:

```
sage: c_dec = C.decode_to_code(r)
sage: c_dec == c
True
```

Perhaps we want the original message back rather than the codeword. All we have to do then is to `unencode` it back to the message space:

```
sage: m_unenc = C.unencode(c_dec)
sage: m_unenc == msg
True
```

It is also possible to perform the two previous operations (correct the errors and recover the original message) in one line, as illustrated below:

```
sage: m_unenc2 = C.decode_to_message(r)
sage: m_unenc2 == msg
True
```
III. A deeper view of the Encoder and Decoder structure

In the previous section, we saw that encoding, decoding and unencoding a vector can be easily done using methods directly on the code object. These methods are actually shortcuts, added for usability, for when one does not care more specifically about how encoding and decoding takes place. At some point, however, one might need more control.

This section will thus go into details on the mechanism of Encoders and Decoders.

At the core, the three mentioned operations are handled by `sage.coding.encoder.Encoder` and `sage.coding.decoder.Decoder`. These objects possess their own methods to operate on words. When one calls (as seen above):

```python
C.encode(msg)
```

one actually calls the method `sage.coding.encoder.Encoder.encode()` on the default encoder of `C`. Every code object possess a list of encoders and decoders it can use. Let us see how one can explore this:

```python
sage: C = codes.GeneralizedReedSolomonCode(GF(59).list()[:40], 12, GF(59).list()[1:41])
sage: C.encoders_available()
['EvaluationPolynomial', 'EvaluationVector', 'Systematic']
sage: C.decoders_available()
['BerlekampWelch', 'ErrorErasure', 'Gao', 'GuruswamiSudan', 'InformationSet', 'KeyEquationSyndrome', 'NearestNeighbor', 'Syndrome']
```

We got a list of the available encoders and decoders for our GRS code. Rather than using the default ones as we did before, we can now ask for specific encoder and decoder:

```python
sage: Evect = C.encoder("EvaluationVector")
sage: Evect
Evaluation vector-style encoder for [40, 12, 29] Generalized Reed-Solomon Code over GF(59)
sage: type(Evect)
<class 'sage.coding.grs_code.GRSEvaluationVectorEncoder'>
sage: msg = random_vector(GF(59), C.dimension())  # random
sage: c = Evect.encode(msg)
sage: NN = C.decoder("NearestNeighbor")
sage: NN
Nearest neighbor decoder for [40, 12, 29] Generalized Reed-Solomon Code over GF(59)
```

Calling:

```python
C.encoder(encoder_name)
```

is actually a short-hand for constructing the encoder manually, by calling the constructor for `sage.coding.grs_code.EncoderGRSEvaluationVector` yourself. If you don't supply `encoder_name` to `sage.coding.linear_code.AbstractLinearCode.encoder()` you get the default encoder for the code. `sage.coding.linear_code.AbstractLinearCode.encoder()` also has an important side-effect: it caches the constructed encoder before returning it. This means that each time one will access the same `EvaluationVector` encoder for `C`, which saves construction time.

All the above things are similar for Decoders. This reinforces that Encoders and Decoders are rarely constructed but
used many times, which allows them to perform expensive precomputation at construction or first use, for the benefit of future use.

This gives a good idea of how the elements work internally. Let us now go a bit more into details on specific points.

### III.1 Message spaces

The point of an Encoder is to encode messages into the code. These messages are often just vectors over the base field of the code and whose length matches the code’s dimension. But it could be anything: vectors over other fields, polynomials, or even something quite different. Therefore, each Encoder has a `sage.coding.encoder.Encoder.message_space()`. For instance, we saw earlier that our GRS code has two possible encoders; let us investigate the one we left behind in the part just before:

```python
sage: Epoly = C.encoder("EvaluationPolynomial")
sage: Epoly
Evaluation polynomial-style encoder for [40, 12, 29] Generalized Reed-Solomon Code over GF(59)
sage: Epoly.message_space()
Univariate Polynomial Ring in x over Finite Field of size 59
sage: msg_p = Epoly.message_space().random_element(degree=C.dimension()-1); msg_p
31*x^11 + 49*x^10 + 56*x^9 + 31*x^8 + 36*x^6 + 58*x^5 + 9*x^4 + 17*x^3 + 29*x^2 + 50*x + 46
```

`Epoly` reflects that GRS codes are often constructed as evaluations of polynomials, and that a natural way to consider their messages is as polynomials of degree at most $k - 1$, where $k$ is the dimension of the code. Notice that the message space of `Epoly` is all univariate polynomials: `message_space` is the ambient space of the messages, and sometimes an Encoder demands that the messages are actually picked from a subspace hereof.

The default encoder of a code always has a vector space as message space, so when we call `sage.coding.linear_code.AbstractLinearCode.decode_to_message()` or `sage.coding.linear_code.AbstractLinearCode.unencode()` on the code itself, as illustrated on the first example, this will always return vectors whose length is the dimension of the code.

### III.2 Generator matrices

Whenever the message space of an Encoder is a vector space and it encodes using a linear map, the Encoder will possess a generator matrix (note that this notion does not make sense for other types of encoders), which specifies that linear map.

Generator matrices have been placed on Encoder objects since a code has many generator matrices, and each of these will encode messages differently. One will also find `sage.coding.linear_code.AbstractLinearCode.generator_matrix()` on code objects, but this is again simply a convenience method which forwards the query to the default encoder.

Let us see this in Sage, using the first encoder we constructed:

```python
sage: Evect.message_space()
Vector space of dimension 12 over Finite Field of size 59
sage: G = Evect.generator_matrix()
sage: G == C.generator_matrix()
True
```
III.3 Decoders and messages

As we saw before, any code has two generic methods for decoding, called decode_to_codeword and decode_to_message. Every Decoder also has these two methods, and the methods on the code simply forward the calls to the default decoder of this code.

There are two reasons for having these two methods: convenience and speed. Convenience is clear: having both methods provides a useful shortcut depending on the user’s needs. Concerning speed, some decoders naturally decode directly to a codeword, while others directly to a message space. Supporting both methods therefore avoids unnecessary work in encoding and unencoding.

However, decode_to_message implies that there is a message space and an encoding from that space to the code behind the scenes. A Decoder has methods message_space and connected_encoder to inform the user about this.

Let us illustrate that by a long example:

```python
sage: C = codes.GeneralizedReedSolomonCode(GF(59).list()[1:41], 3, GF(59).list()[1:41])
sage: c = C.random_element()
sage: c
True

#Create two decoders: Syndrome and Gao
sage: Syn = C.decoder("KeyEquationSyndrome")
sage: Gao = C.decoder("Gao")

#Check their message spaces
sage: Syn.message_space()
Vector space of dimension 3 over Finite Field of size 59
sage: Gao.message_space()
Univariate Polynomial Ring in x over Finite Field of size 59

#and now we unencode
sage: Syn.decode_to_message(c) #random
(55,9,43)
sage: Gao.decode_to_message(c) #random
43*x^2 + 9*x + 55
```

IV. A deeper look at channels

In Section II, we briefly introduced the Channel objects as a way to put errors in a word. In this section, we will look deeper at their functionality and introduce a second Channel.

Note: Once again, we chose a specific class as a running example through all this section, as we do not want to make an exhaustive catalog of all channels. If one wants to get this list, one can access it by typing:

```python
channels.<tab>
```

in Sage.

Consider again the sage.coding.channel.ChannelStaticErrorRate() from before. This is a channel that places errors in the transmitted vector but within controlled boundaries. We can describe these boundaries in two ways:

- The first one was illustrated in Section II and consists in passing an integer, as shown below:
We can also pass a tuple of two integers, the first smaller than the second. Then each time a word is transmitted, a random number of errors between these two integers will be added:

```
sage: t = (1, 14)
sage: Chan = channels.StaticErrorRateChannel(C.ambient_space(), t)
sage: Chan
Static error rate channel creating between 1 and 14 errors, of input and output space Vector space of dimension 40 over Finite Field of size 59
```

We already know that a channel has a `sage.coding.channel.Channel.transmit()` method which will perform transmission over the channel; in this case it will return the transmitted word with some errors in it. This method will always check if the provided word belongs to the input space of the channel. In a case one is absolutely certain that one’s word is in the input space, one might want to avoid this check, which is time consuming - especially if one is simulating millions of transmissions. For this usage there is `sage.coding.channel.Channel.transmit_unsafe()` which does the same as `sage.coding.channel.Channel.transmit()` but without checking the input, as illustrated thereafter:

```
sage: c = C.random_element()
sage: c in C
True
sage: c_trans = Chan.transmit_unsafe(c)
sage: c_trans in C
False
```

Note there exists a useful shortcut for `sage.coding.channel.Channel.transmit()`

```
sage: r = Chan(c)
sage: r in C
False
```

**A channel for errors and erasures**

Let us introduce a new Channel object which adds errors and erasures. When it transmits a word, it both adds some errors as well as it erases some positions:

```
sage: Chan = channels.ErrorErasureChannel(C.ambient_space(), 3, 4)
sage: Chan
Error-and-erasure channel creating 3 errors and 4 erasures of input space Vector space of dimension 40 over Finite Field of size 59 and output space The Cartesian product of (Vector space of dimension 40 over Finite Field of size 59, Vector space of dimension 40 over Finite Field of size 2)
```

The first parameter is the input space of the channel. The next two are (respectively) the number of errors and the number of erasures. Each of these can be tuples too, just as it was with `sage.coding.channel.StaticErrorRateChannel`. As opposed to this channel though, the output of `sage.coding.channel.ErrorErasureChannel` is not the same as its input space, i.e. the ambient space of C. Rather, it will return two vectors: the first is the transmitted word with the
errors added and erased positions set to 0. The second one is the erasure vector whose erased positions contain ones. This is reflected in `sage.coding.channel.output_space()`:

```python
sage: C = codes.random_linear_code(GF(7), 10, 5)
sage: Chan.output_space()
The Cartesian product of (Vector space of dimension 40 over Finite Field of size 59, Vector space of dimension 40 over Finite Field of size 2)
sage: Chan(c) # random
((0, 3, 6, 4, 0, 0, 0, 1),
(1, 0, 0, 0, 1, 0, 0, 1))
```

Note it is guaranteed by construction that errors and erasures will never overlap, so when you ask for \( e \) errors and \( t \) erasures, you will always receive a vector with \( e \) errors and \( t \) erased positions.

### V. Codes in General

So far we have talked only about codes which are linear and over the Hamming metric. Sage also supports codes which are non-linear and/or over a different metric. Since working with these usually involves creating new classes, they are covered in detail in *How to write your own classes for coding theory*.

For this tutorial, we include a small example of a linear code over the rank metric. Unlike the Hamming metric, where the distance between two words is the number of positions in which their differ, rank metric takes distance to be the rank of the difference of two codewords, which are represented as matrices.

We are going to create a generic linear code over the rank metric without any additional structure. To build such a code, we only need a generator matrix:

```python
sage: G = Matrix(GF(4), [[1,1,0], [0,0,1]])
sage: C = codes.LinearRankMetricCode(G)
```

We can do all the things that we did with our example of a linear code over the Hamming metric. Therefore we are going to focus on the different metric.

Take a word in our code:

```python
sage: c = C[1]
sage: c
(1, 1, 0)
```

Over the usual Hamming metric, the weight of this word would be 2. However, over the rank metric, we get a different result:

```python
sage: C.rank_weight_of_vector(c)
1
```

The weight of a word in the rank metric is simply the rank of the matrix form of the word:

```python
sage: C.matrix_form_of_vector(c)
[1 1 0]
[0 0 0]
```

As we said before, the distance between two words is the rank of their difference:

```python
sage: d = C[2]
sage: d
```
(continued from previous page)

Even though the words \( c \) and \( d \) differ in two positions, their distance over the rank metric is 1:

```python
sage: C.rank_distance_between_vectors(c, d)
1
```

For more details on the linear rank metric code, see Generic structures for linear codes over the rank metric.

**VI. Conclusion - Afterword**

This last section concludes our tutorial on coding theory.

After reading this, you should know enough to create and manipulate codes in Sage!

We did not illustrate all the content of the library in this tutorial. For instance, we did not mention how Sage manages bounds on codes.

All objects, constructions and methods related to coding theory are hidden under the prefix `codes` in Sage.

For instance, it is possible to find all encoders you can build by typing:

```python
codes.encoders.<tab>
```

So, if you are looking for a specific object related to code, you should always type:

```python
codes.<tab>
```

and check if there’s a subcategory which matches your needs.

Despite all the hard work we put on it, there’s always much to do!

Maybe at some point you might want to create your own codes for Sage. If it’s the case and if you don’t know how to do that, don’t panic! We also wrote a tutorial for this specific case, which you can find here: *How to write your own classes for coding theory*.

### 12.1.9 How to write your own classes for coding theory

*Author: David Lucas (2016) for initial version. Marketa Slukova (2019) for this version.*

This tutorial, designed for advanced users who want to build their own classes, will explain step by step what you need to do to write code which integrates well in the framework of coding theory. During this tutorial, we will cover the following parts:

- how to write a new **code family**
- how to write a new **encoder**
- how to write a new **decoder**
- how to write a new **channel**

This tutorial focuses on the example of implementation of the repetition code. At the end of each part, we will summarize every important step of the implementation. If one just wants a quick access to the implementation of one of the objects cited above, one can jump directly to the end of related part, which presents a summary of what to do.
<table>
<thead>
<tr>
<th>I. Abstract classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>There is a number of abstract classes representing different types of codes in the coding module of Sage. Depending on the type of code you want to implement, you should inherit from different abstract classes.</td>
</tr>
<tr>
<td>The most generic class is <code>sage.coding.abstract_code:AbstractCode</code>. This class makes no assumptions about linearity, metric, finiteness or the number of alphabets. The abstract notion of “code” that is implicitly used for this class is any enumerable subset of a cartesian product $A_1 \times A_2 \times \ldots \times A_n$ for some sets $A_i$.</td>
</tr>
<tr>
<td>If, for example, you want to create a non-linear code family, this is the class you should inherit from.</td>
</tr>
<tr>
<td>We give a small example of creating a non-linear code family. Take the code consisting of the following $l$ words: ${00\ldots00, 10\ldots00, 11\ldots00, \ldots, 11\ldots10, 11\ldots11}$.</td>
</tr>
<tr>
<td>Here is how we can implement it:</td>
</tr>
<tr>
<td>```python</td>
</tr>
<tr>
<td>sage: from sage.coding.abstract_code import AbstractCode</td>
</tr>
<tr>
<td>sage: class ExampleCodeFamily(AbstractCode):</td>
</tr>
<tr>
<td>....: def <strong>init</strong>(self, length):</td>
</tr>
<tr>
<td>....: super(ExampleCodeFamily, self).<strong>init</strong>(length)</td>
</tr>
<tr>
<td>....: def <strong>iter</strong>(self):</td>
</tr>
<tr>
<td>....: for i in range(self.length() + 1):</td>
</tr>
<tr>
<td>....: yield vector([1 for j in range(i)] + [0 for k in range(i, self.length())])</td>
</tr>
<tr>
<td>....: def <strong>contains</strong>(self, word):</td>
</tr>
<tr>
<td>....: return word in list(self)</td>
</tr>
<tr>
<td>....: def <strong>repr</strong>(self):</td>
</tr>
<tr>
<td>....: return &quot;Dummy code of length {}&quot;.format(self.length())</td>
</tr>
<tr>
<td>sage: C = ExampleCodeFamily(4) # check that this works.</td>
</tr>
<tr>
<td>sage: C.list()</td>
</tr>
<tr>
<td>[(0, 0, 0, 0), (1, 0, 0, 0), (1, 1, 0, 0), (1, 1, 1, 0), (1, 1, 1, 1)]</td>
</tr>
<tr>
<td>Focusing on linear codes, the most generic representative is the class <code>sage.coding.linear_code_no_metric:AbstractLinearCodeNoMetric</code> which contains all the methods that all linear codes share regardless of what their metric is. If you want to implement a linear code over some metric which has not been implemented in Sage yet, this is the class you should be inheriting from.</td>
</tr>
</tbody>
</table>
We have two metric specific abstract linear code classes, `sage.coding.linear_code:AbstractLinearCode` for Hamming metric and `sage.coding.linear_rank_metric:AbstractLinearRankMetricCode` for rank metric. If you wish to implement a linear code class over one of these metrics, you should inherit from the given abstract class.

II. The repetition code

We want to implement in Sage the well-known repetition code. Its definition follows: the \((n, 1)\)-repetition code over \(F_q\) is the code formed by all the vectors of \(F_q^n\) of the form \((i, i, ... , i)\) for all \(i \in F_q\).

For example, the \((3, 1)\)-repetition code over \(F_2\) is: \(C = \{(0, 0, 0), (1, 1, 1)\}\).

The encoding is very simple, it only consists in repeating \(n\) times the input symbol and pick the vector thus formed.

The decoding uses majority voting to select the right symbol (over \(F_2\)). If we receive the word \((1, 0, 1)\) (example cont’d), we deduce that the original word was \((1)\). It can correct up to \(\left\lfloor \frac{n-1}{2} \right\rfloor\) errors.

Through all this tutorial, we will illustrate the implementation of the \((n, 1)\)-repetition code over \(F_2\).

III. Write a new code class

The first thing to do to write a new code class is to identify which abstract class to inherit from. Since the repetition code is linear and we take it over the Hamming metric, this means that we will inherit from `sage.coding.linear_code.AbstractLinearCode`.

Now we have to identify the initializing parameters of this class, which are:

- the length of the code,
- the base field of the code,
- the default encoder for the code,
- the default decoder for the code and
- any other useful argument we want to set at construction time.

For our code, we know its length, its dimension, its base field, one encoder and one decoder.

Now we isolated the parameters of the code, we can write the constructor of our class. As we said, every linear code class over the Hamming metric must inherit from `sage.coding.linear_code.AbstractLinearCode`. This class provides a lot of useful methods and, as we illustrate thereafter, a default constructor which sets the length, the base field, the default encoder and the default decoder as class parameters. We also need to create the dictionary of known encoders and decoders for the class.

Let us now write the constructor for our code class, that we store in some file called `repetition_code.py`:

```python
sage: from sage.coding.linear_code import AbstractLinearCode
sage: from sage.rings.finite_rings.finite_field_constructor import FiniteField as GF
sage: class BinaryRepetitionCode(AbstractLinearCode):
    ....:     _registered_encoders = {}
    ....:     _registered_decoders = {}
    ....:     def __init__(self, length):
    ....:         super(BinaryRepetitionCode, self).__init__(GF(2), length,
    ....:                                           "RepetitionGeneratorMatrixEncoder", "MajorityVoteDecoder")
    ....:         self._dimension = 1
```

As you notice, the constructor is really simple. Most of the work is indeed managed by the topclass through the `super` statement. Note that the dimension is not set by the abstract class, because for some code families the exact dimension is hard to compute. If the exact dimension is known, set it using `_dimension` as a class parameter.
We can now write representation methods for our code class:

```
sage: def _repr_(self):
....:     return "Binary repetition code of length %s" % self.length()

sage: def _latex_(self):
....:     return "\textnormal{Binary repetition code of length } %s" % self.length()
```

We also write a method to check equality:

```
sage: def __eq__(self, other):
....:     return (isinstance(other, BinaryRepetitionCode)
....:             and self.length() == other.length()
....:             and self.dimension() == other.dimension())
```

After these examples, you probably noticed that we use two methods, namely `length()` and `dimension()` without defining them. That is because their implementation is provided in parent classes of `sage.coding.linear_code.AbstractLinearCode`, which are `sage.coding.linear_code_no_metric.AbstractLinearCodeNoMetric` and `sage.coding.abstract_code.AbstractCode`. They provide a default implementation of the following getter methods:

- `sage.coding.abstract_code.AbstractCode.length()`,
- `sage.coding.linear_code_no_metric.AbstractLinearCodeNoMetric.dimension()`,
- `sage.coding.linear_code_no_metric.AbstractLinearCodeNoMetric.base_field()` and
- `sage.coding.linear_code_no_metric.AbstractLinearCodeNoMetric.ambient_space()`.

They also provide several other useful methods, such as `__contains__`. Note that a lot of these other methods rely on the computation of a generator matrix. It is thus highly recommended to set an encoder which knows how to compute such a matrix as default encoder. As default encoder will be used by all these methods which expect a generator matrix, if one provides a default encoder which does not have a `generator_matrix` method, a lot of generic methods will fail.

As our code family is really simple, we do not need anything else, and the code provided above is enough to describe properly a repetition code.

**Summary of the implementation for linear codes over the Hamming metric**

1. Inherit from `sage.coding.linear_code.AbstractLinearCode`.

2. Add `_registered_encoders = {}` and `_registered_decoders = {}` as class variables.

3. Add this line in the class’ constructor:

   ```python
   super(ClassName, self).__init__(base_field, length, "DefaultEncoder",
   "DefaultDecoder")
   ```

4. Implement representation methods (not mandatory, but highly advised) `_repr_` and `_latex_`.

5. Implement `__eq__`.

6. `__ne__`, `length` and `dimension` come with the abstract class.

Please note that `dimension` will not work if there is no field `_dimension` as class parameter.

We now know how to write a new code class. Let us see how to write a new encoder and a new decoder.
IV. Write a new encoder class

Let us continue our example. We ask the same question as before: what do we need to describe the encoder? For most of the cases (this one included), we only need the associated code. In that case, writing the constructor is really straightforward (we store the code in the same .py file as the code class):

```python
sage: from sage.coding.encoder import Encoder
sage: class BinaryRepetitionCodeGeneratorMatrixEncoder(Encoder):
    ....:     def __init__(self, code):
    ....:         super(BinaryRepetitionCodeGeneratorMatrixEncoder, self).__init__(code)
```

Same thing as before, as an encoder always needs to know its associated code, the work can be done by the base class. Remember to inherit from `sage.coding.encoder.Encoder`!

We also want to override representation methods `_repr_` and `_latex_`:

```python
sage: def _repr_(self):
    ....:     return "Binary repetition encoder for the %s" % self.code()

sage: def _latex_(self):
    ....:     return "\textnormal{Binary repetition encoder for the } %s" % self.code()
```

And we want to have an equality check too:

```python
sage: def __eq__(self, other):
    ....:     return (isinstance(other, BinaryRepetitionCodeGeneratorMatrixEncoder)
    ....:             and self.code() == other.code())
```

As before, default getter method is provided by the top class, namely `sage.coding.encoder.Encoder.code()`.

All we have to do is to implement the methods related to the encoding. This implementation changes quite a lot whether we have a generator matrix or not.

We have a generator matrix

In that case, the message space is a vector space, and it is especially easy: the only method you need to implement is `generator_matrix`.

Continuing our example, it will be:

```python
sage: def generator_matrix(self):
    ....:     n = self.code().length()
    ....:     return Matrix(GF(2), 1, n, [GF(2).one()]*n)
```

As the topclass provides default implementation for encode and the inverse operation, that we call `unencode` (see: `sage.coding.encoder.Encoder.encode()` and `sage.coding.encoder.Encoder.unencode()`), alongside with a default implementation of `sage.coding.encoder.Encoder.message_space()`, our work here is done.

Note: Default encode method multiplies the provide word by the generator matrix, while default `unencode` computes an information set for the generator matrix, inverses it and performs a matrix-vector multiplication to recover the original message. If one has a better implementation for one’s specific code family, one should obviously override the default encode and unencode.
We do not have any generator matrix

In that case, we need to override several methods, namely encode, unencode_nocheck and probably message_space (in the case where the message space is not a vector space). Note that the default implementation of sage.coding.encoder.Encoder.unencode() relies on unencode_nocheck, so reimplementing the former is not necessary.

In our example, it is easy to create an encoder which does not need a generator matrix to perform the encoding and the unencoding. We propose the following implementation:

```python
sage: def encode(self, message):
    ....:     return vector(GF(2), [message] * self.code().length())

sage: def unencode_nocheck(self, word):
    ....:     return word[0]

sage: def message_space(self):
    ....:     return GF(2)
```

Our work here is done.

We need to do one extra thing: set this encoder in the dictionary of known encoders for the associated code class. To do that, just add the following line at the end of your file:

```python
BinaryRepetitionCode._registered_encoders["RepetitionGeneratorMatrixEncoder"] = BinaryRepetitionCodeGeneratorMatrixEncoder
```

Note: In case you are implementing a generic encoder (an encoder which works with any family of linear codes), please add the following statement in AbstractLinearCode's constructor instead: self._registered_encoders["EncName"] = MyGenericEncoder. This will make it immediately available to any code class which inherits from AbstractLinearCode.

Summary of the implementation for encoders

1. Inherit from sage.coding.encoder.Encoder.
2. Add this line in the class' constructor:
   ```python
   super(ClassName, self).__init__(associated_code)
   ```
3. Implement representation methods (not mandatory) _repr_ and _latex_.
4. Implement __eq__
5. __ne__, code come with the abstract class.
6. If a generator matrix is known, override generator_matrix.
7. Else override encode, unencode_nocheck and if needed message_space.
8. Add the encoder to CodeClass._registered_encoders.
V. Write a new decoder class

Let us continue by writing a decoder. As before, we need to know what is required to describe a decoder. We need of course the associated code of the decoder. We also want to know which Encoder we should use when we try to recover the original message from a received word containing errors. We call this encoder connected_encoder. As different decoding algorithms do not have the same behaviour (e.g. probabilistic decoding vs deterministic), we would like to give a few clues about the type of a decoder. So we can store a list of keywords in the class parameter _decoder_type. Eventually, we also need to know the input space of the decoder. As usual, initializing these parameters can be delegated to the topclass, and our constructor looks like that:

```python
sage: from sage.coding.decoder import Decoder
sage: class BinaryRepetitionCodeMajorityVoteDecoder(Decoder):
....:     def __init__(self, code):
....:         super((BinaryRepetitionCodeMajorityVoteDecoder, self).__init__(code,
....:         code.ambient_space(), "RepetitionGeneratorMatrixEncoder"))
```

Remember to inherit from `sage.coding.decoder.Decoder`!

As _decoder_type is actually a class parameter, one should set it in the file itself, outside of any method. For readability, we suggest to add this statement at the bottom of the file. We’ll get back to this in a moment.

We also want to override representation methods _repr_ and _latex_:

```python
sage: def __repr__(self):
....:     return "Majority vote-based decoder for the %s" % self.code()
```

```python
sage: def __latex__(self):
....:     return "\textnormal{Majority vote-based decoder for the } %s" % self.code()
```

And we want to have an equality check too:

```python
sage: def __eq__(self, other):
....:     return isinstance((other, BinaryRepetitionCodeMajorityVoteDecoder)
....:     and self.code() == other.code())
```

As before, default getter methods are provided by the topclass, namely `sage.coding.decoder.Decoder.code()`, `sage.coding.decoder.Decoder.input_space()`, `sage.coding.decoder.Decoder.decoder_type()` and `sage.coding.decoder.Decoder.connected_encoder()`.

All we have to do know is to implement the methods related to the decoding.

There are two methods, namely `sage.coding.decoder.Decoder.decode_to_code()` and `sage.coding.decoder.Decoder.decode_to_message()`.

By the magic of default implementation, these two are linked, as `decode_to_message` calls first `decode_to_code` and then `unencode`, while `decode_to_code` calls successively `decode_to_message` and `encode`. So we only need to implement one of these two, and we choose to override `decode_to_code`:

```python
sage: def decode_to_code(self, word):
....:     list_word = word.list()
....:     count_one = list_word.count(GF(2).one())
....:     n = self.code().length()
....:     length = len(list_word)
....:     F = GF(2)
....:     if count_one > length / 2:
....:         return vector(F, [F.one()] * n)
....:     elif count_one < length / 2:
```

(continues on next page)
.....: return vector(F, [F.zero()] * n)
.....: else:
.....: raise DecodingError("impossible to find a majority")

Note: One notices that if default decode_to_code calls default decode_to_message and default decode_to_message calls default decode_to_code, if none is overridden and one is called, it will end up stuck in an infinite loop. We added a trigger guard against this, so if none is overridden and one is called, an exception will be raised.

Only one method is missing: one to provide to the user the number of errors our decoder can decode. This is the method sage.coding.decoder.Decoder.decoding_radius(), which we override:

```python
sage: def decoding_radius(self):
.....: return (self.code().length()-1) // 2
```

As for some cases, the decoding might not be precisely known, its implementation is not mandatory in sage.coding.decoder.Decoder's subclasses.

We need to do one extra thing: set this encoder in the dictionary of known decoders for the associated code class. To do that, just add the following line at the end of your file:

```python
BinaryRepetitionCode._registered_decoders["MajorityVoteDecoder"] = BinaryRepetitionCodeMajorityVoteDecoder
```

Also put this line to set decoder_type:

```python
BinaryRepetitionCode._decoder_type = {"hard-decision", "unique"}
```

Note: In case you are implementing a generic decoder (a decoder which works with any family of linear codes), please add the following statement in AbstractLinearCode's constructor instead: self._registered_decoders["DecName"] = MyGenericDecoder. This will make it immediately available to any code class which inherits from AbstractLinearCode.

### Summary of the implementation for decoders

1. Inherit from sage.coding.decoder.Decoder.
2. Add this line in the class' constructor:
   ```python
   super(ClassName, self).__init__(associated_code, input_space, connected_encoder_name, decoder_type)
   ```
3. Implement representation methods (not mandatory) _repr_ and _latex_.
4. Implement __eq__.
5. __ne__, code, connected_encoder, decoder_type come with the abstract class.
6. Override decode_to_code or decode_to_message and decoding_radius.
7. Add the encoder to CodeClass._registered_decoders.
VI. Write a new channel class

Alongside all these new structures directly related to codes, we also propose a whole new and shiny structure to experiment on codes, and more specifically on their decoding.

Indeed, we implemented a structure to emulate real-world communication channels. I’ll propose here a step-by-step implementation of a dummy channel for example’s sake.

We will implement a very naive channel which works only for words over $\mathbb{F}_2$ and flips as many bits as requested by the user.

As channels are not directly related to code families, but more to vectors and words, we have a specific file, `channel.py` to store them.

So we will just add our new class in this file.

For starters, we ask ourselves the eternal question: What do we need to describe a channel? Well, we mandatorily need its `input_space` and its `output_space`. Of course, in most of the cases, the user will be able to provide some extra information on the channel’s behaviour. In our case, it will be the number of bits to flip (aka the number of errors).

As you might have guess, there is an abstract class to take care of the mandatory arguments! Plus, in our case, as this channel only works for vectors over $\mathbb{F}_2$, the input and output spaces are the same. Let us write the constructor of our new channel class:

```python
from sage.coding.channel import Channel
class BinaryStaticErrorRateChannel(Channel):
    def __init__(self, space, number_errors):
        if space.base_ring() is not GF(2):
            raise ValueError("Provided space must be a vector space over GF(2)")
        if number_errors > space.dimension():
            raise ValueError("number_errors cannot be bigger than input space's dimension")
        super(BinaryStaticErrorRateChannel, self).__init__(space, space)
        self._number_errors = number_errors

sage: from sage.coding.channel import Channel
sage: class BinaryStaticErrorRateChannel(Channel):
    ....:    def __init__(self, space, number_errors):
    ....:        if space.base_ring() is not GF(2):
    ....:            raise ValueError("Provided space must be a vector space over GF(2)")
    ....:        if number_errors > space.dimension():
    ....:            raise ValueError("number_errors cannot be bigger than input space's dimension")
    ....:        super(BinaryStaticErrorRateChannel, self).__init__(space, space)
    ....:        self._number_errors = number_errors
```

Remember to inherit from `sage.coding.channel.Channel`!

We also want to override representation methods `_repr_` and `_latex_`:

```python
sage: def _repr_(self):
    ....:    return "Binary static error rate channel creating %s errors, of input and output space %s"
    ....:    % (format_interval(no_err), self.input_space())

sage: def _latex_(self):
    ....:    return "\textnormal{Static error rate channel creating %s errors, of input and output space %s}"% (format_interval(no_err), self.input_space())
```

We don’t really see any use case for equality methods (`__eq__` and `__ne__`) so do not provide any default implementation. If one needs these, one can of course override Python’s default methods.

We of course want getter methods. There is a provided default implementation for `input_space` and `output_space`, so we only need one for `number_errors`:

```python
sage: def number_errors(self):
    ....:    return self._number_errors
```

12.1. Thematic tutorial document tree
So, now we want a method to actually add errors to words. As it is the same thing as transmitting messages over a real-world channel, we propose two methods, `transmit` and `transmit_unsafe`. As you can guess, `transmit_unsafe` tries to transmit the message without checking if it is in the input space or not, while `transmit` checks this before the transmission... Which means that `transmit` has a default implementation which calls `transmit_unsafe`. So we only need to override `transmit_unsafe`! Let us do it:

```python
sage: def transmit_unsafe(self, message):
    w = copy(message)
    number_err = self.number_errors()
    V = self.input_space()
    F = GF(2)
    for i in sample(range(V.dimension()), number_err):
        w[i] += F.one()
    return w
```

That is it, we now have our new channel class ready to use!

### Summary of the implementation for channels

1. Inherit from `sage.coding.channel.Channel`.
2. Add this line in the class' constructor:
   ```python
   super(ClassName, self).__init__(input_space, output_space)
   ```
3. Implement representation methods (not mandatory) `__repr__` and `__latex__`.
4. `input_space` and `output_space` getter methods come with the abstract class.
5. Override `transmit_unsafe`.

### VII. Sort our new elements

As there is many code families and channels in the coding theory library, we do not wish to store all our classes directly in Sage’s global namespace.

We propose several catalog files to store our constructions, namely:

- `codes_catalog.py`
- `encoders_catalog.py`
- `decoders_catalog.py`
- `channels_catalog.py`

Every time one creates a new object, it should be added in the dedicated catalog file instead of coding theory folder’s `all.py`

Here it means the following:

- add the following in `codes_catalog.py`:
  ```python
  from sage.coding.repetition_code import BinaryRepetitionCode
  ```

- add the following in `encoders_catalog.py`:
from sage.coding.repetition_code import...
   BinaryRepetitionCodeGeneratorMatrixEncoder

• add the following in decoders_catalog.py:
from sage.coding.repetition_code import...
   BinaryRepetitionCodeMajorityVoteDecoder

• add the following in channels_catalog.py:
from sage.coding.channel import BinaryStaticErrorRateChannel

VIII. Complete code of this tutorial

If you need some base code to start from, feel free to copy-paste and derive from the one that follows.

repetition_code.py (with two encoders):

```python
from sage.coding.linear_code import AbstractLinearCode
from sage.coding.encoder import Encoder
from sage.coding.decoder import Decoder
from sage.rings.finite_rings.finite_field_constructor import FiniteField as GF

class BinaryRepetitionCode(AbstractLinearCode):
    _registered_encoders = {}
    _registered_decoders = {}

    def __init__(self, length):
        super(BinaryRepetitionCode, self).__init__(GF(2), length,
            "RepetitionGeneratorMatrixEncoder", "MajorityVoteDecoder")
        self._dimension = 1

    def __repr__(self):
        return "Binary repetition code of length %s" % self.length()

    def _latex_(self):
        return "\textnormal{Binary repetition code of length } %s" % self.length()

    def __eq__(self, other):
        return (isinstance(other, BinaryRepetitionCode) 
            and self.length() == other.length()
            and self.dimension() == other.dimension())

class BinaryRepetitionCodeGeneratorMatrixEncoder(Encoder):
    def __init__(self, code):
        super(BinaryRepetitionCodeGeneratorMatrixEncoder, self).__init__(code)

    def __repr__(self):
        ...
```

(continues on next page)
return "Binary repetition encoder for the %s" % self.code()

def _latex_(self):
    return r"\textnormal{Binary repetition encoder for the } %s" % self.code()

def __eq__(self, other):
    return (isinstance(other, BinaryRepetitionCodeGeneratorMatrixEncoder)
    and self.code() == other.code())

def generator_matrix(self):
    n = self.code().length()
    return Matrix(GF(2), 1, n, [GF(2).one()] * n)

class BinaryRepetitionCodeStraightforwardEncoder(Encoder):
    def __init__(self, code):
        super(BinaryRepetitionCodeStraightforwardEncoder, self).__init__(code)
    
    def _repr_(self):
        return "Binary repetition encoder for the %s" % self.code()
    
    def _latex_(self):
        return r"\textnormal{Binary repetition encoder for the } %s" % self.code()
    
    def __eq__(self, other):
        return (isinstance(other, BinaryRepetitionCodeStraightforwardEncoder)
        and self.code() == other.code())
    
    def encode(self, message):
        return vector(GF(2), [message] * self.code().length())
    
    def unencode_nocheck(self, word):
        return word[0]
    
    def message_space(self):
        return GF(2)

class BinaryRepetitionCodeMajorityVoteDecoder(Decoder):
    def __init__(self, code):
        super(BinaryRepetitionCodeMajorityVoteDecoder, self).__init__(code, code.ambient_ → space(), "RepetitionGeneratorMatrixEncoder")
    
    def _repr_(self):
        return "Majority vote-based decoder for the %s" % self.code()
    
    def _latex_(self):
\textnormal{ Majority vote based-decoder for the } %s % s % self.code()

\begin{Verbatim}
\begin{verbatim}
def \_eq\_(self, other):
    return (isinstance(other, BinaryRepetitionCodeMajorityVoteDecoder)
        and self.code() == other.code())

def decode_to_code(self, word):
    list_word = word.list()
    count_one = list_word.count(GF(2).one())
    n = self.code().length()
    length = len(list_word)
    F = GF(2)
    if count_one > length / 2:
        return vector(F, [F.one()] * n)
    elif count_one < length / 2:
        return vector(F, [F.zero()] * n)
    else:
        raise DecodingError("impossible to find a majority")

def decoding_radius(self):
    return (self.code().length()-1) // 2
\end{verbatim}
\end{Verbatim}

BinaryRepetitionCode._registered_encoders["RepetitionGeneratorMatrixEncoder"] = BinaryRepetitionCodeGeneratorMatrixEncoder
BinaryRepetitionCode._registered_encoders["RepetitionStraightforwardEncoder"] = BinaryRepetitionCodeStraightforwardEncoder
BinaryRepetitionCode._registered_decoders["MajorityVoteDecoder"] = BinaryRepetitionCodeMajorityVoteDecoder
BinaryRepetitionCodeMajorityVoteDecoder._decoder_type = {"hard-decision", "unique"}

channel.py (continued):

\begin{Verbatim}
\begin{verbatim}
class BinaryStaticErrorRateChannel(Channel):
    def \_init\_(self, space, number_errors):
        if space.base_ring() is not GF(2):
            raise ValueError("Provided space must be a vector space over GF(2)")
        if number_errors > space.dimension():
            raise ValueErrors("number_errors cannot be bigger than input space's dimension")
        super(BinaryStaticErrorRateChannel, self)._init_(space, space)
        self._number_errors = number_errors

    def \_repr\_(self):
        return ("Binary static error rate channel creating %s errors, of input and output space %s"
            % (format_interval(no_err), self.input_space()))

    def \_latex\_(self):\end{verbatim}
\end{Verbatim}
Thematic Tutorials, Release 9.7

(continued from previous page)

return \textnormal{\textbackslash
Static error rate channel creating %s errors, of input and output space %s}"
% (format_interval(no_err), self.input_space())

def number_errors(self):
    return self._number_errors

def transmit_unsafe(self, message):
    w = copy(message)
    number_err = self.number_errors()
    V = self.input_space()
    F = GF(2)
    for i in sample(range(V.dimension()), number_err):
        w[i] += F.one()
    return w

codes_catalog.py (continued):

from sage.coding.repetition_code import BinaryRepetitionCode

encoders_catalog.py (continued):

from sage.coding.repetition_code import (BinaryRepetitionCodeGeneratorMatrixEncoder,
                                          BinaryRepetitionCodeStraightforwardEncoder)

decoders_catalog.py (continued):

from sage.coding.repetition_code import BinaryRepetitionCodeMajorityVoteDecoder

channels_catalog.py (continued):

from sage.coding.channel import (ErrorErasureChannel, StaticErrorRateChannel,
                                BinaryStaticErrorRateChannel)

12.1.10 Polyhedra

Here you can find various documents that explain how to perform polyhedral computations in Sage.

A Brief Introduction to Polytopes in Sage

Author: sarah-marie belcastro <smbelcas@toroidalsnark.net>

If you already know some convex geometry \textit{a la} Grünbaum or Brøndsted, then you may have itched to get your hands dirty with some polytope calculations. Here is a mini-guide to doing just that.
Basics

First, let’s define a polytope as the convex hull of a set of points, i.e. given $S$ we compute $P = \text{conv}(S)$:

```
 sage: P1 = Polyhedron(vertices = [[-5,2], [4,4], [3,0], [1,0], [2,-4], [-3,-1], [-5,-3]])
 sage: P1
 A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 4 vertices
```

Notice that Sage tells you the dimension of the polytope as well as the dimension of the ambient space.

Of course, you want to know what this object looks like:

```
 sage: P1.plot()
 Graphics object consisting of 6 graphics primitives
```

Even in only 2 dimensions, it’s a pain to figure out what the supporting hyperplanes are. Luckily Sage will take care of that for us.

```
 sage: for q in P1.Hrepresentation():
 ....:     print(q)
 An inequality (-4, 1) x + 12 >= 0
 An inequality (1, 7) x + 26 >= 0
 An inequality (1, 0) x + 5 >= 0
 An inequality (2, -9) x + 28 >= 0
```

That notation is not immediately parseable, because seriously, those do not look like equations of lines (or of halfspaces, which is really what they are).

$(-4, 1) x + 12 \geq 0$ really means $(-4,1) \cdot \vec{x} + 12 \geq 0$.

So... if you want to define a polytope via inequalities, you have to translate each inequality into a vector. For example, $(-4,1) \cdot \vec{x} + 12 \geq 0$ becomes $(12, -4, 1)$.

```
 sage: altP1 = Polyhedron(ieqs=[(12, -4, 1), (26, 1, 7),(5,1,0), (28, 2, -9)])
 sage: altP1.plot()
 Graphics object consisting of 6 graphics primitives
```

Other information you might want to pull out of Sage about a polytope is the vertex list, which can be done in two ways:

```
 sage: for q in P1.Vrepresentation():
 ....:     print(q)
 A vertex at (-5, -3)
 A vertex at (-5, 2)
 A vertex at (4, 4)
 A vertex at (2, -4)
```

```
 sage: P1.vertices()
 (A vertex at (-5, -3), A vertex at (-5, 2), A vertex at (4, 4), A vertex at (2, -4))
```
Polar duals

Surely you want to compute the polar dual:

```
sage: P1dual = P1.polar()
sage: P1dual
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 4 vertices
```

Check it out---we started with an integer-lattice polytope and dualized to a rational-lattice polytope. Let’s look at that.

```
sage: P1dual.plot()
Graphics object consisting of 6 graphics primitives
```

```
sage: P1.plot() + P1dual.plot()
Graphics object consisting of 12 graphics primitives
```

Oh, yeah, unless the polytope is unit-sphere-sized, the dual will be a very different size. Let’s rescale.

```
sage: ((1/4)*P1).plot() + (4*P1dual).plot()
Graphics object consisting of 12 graphics primitives
```

If you think that looks a little bit shady, you’re correct. Here is an example that makes the issue a bit clearer.

```
sage: P2 = Polyhedron(vertices = [[-5,0], [-1,1], [-2,0], [1,0], [-2,-1], [-3,-1], [-5,-1]])
sage: P2
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 5 vertices
sage: P2dual = P2.polar(); P2dual
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 5 vertices
sage: P2.plot() + P2dual.plot()
Graphics object consisting of 14 graphics primitives
```

That is clearly not computing what we think of as the polar dual. But look at this...

```
sage: P2.plot() + (-1*P2dual).plot()
Graphics object consisting of 14 graphics primitives
```

Here is what’s going on.

If a polytope $P$ is in $\mathbb{Z}$, then...

(1) … the dual is inverted in some way, which is vertically for polygons.

(2) … the dual is taken of $P$ itself.

(3) … if the origin is not in $P$, then an error is returned.

However, if a polytope is not in $\mathbb{Z}$, for example if it’s in $\mathbb{Q}$ or $\mathbb{R}D$, then...

(1’) … the dual is not inverted.

(2’) … the dual is taken of $P$-translated-so-barycenter-is-at-origin.

Keep all of this in mind as you take polar duals.
Polytope Constructions

Minkowski sums! Now with two syntaxes!

```sage
P1+P2
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 8 vertices
```

```sage
P1.minkowski_sum(P2)
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 8 vertices
```

Okay, fine. We should have some 3-dimensional examples, at least. (Note that in order to display polytopes effectively you’ll need visualization software such as Javaview and Jmol installed.)

```sage
P3 = Polyhedron(vertices=[(0,0,0), (0,0,1/2), (0,1/2,0), (1/2,0,0), (3/4,1/5,3/2)]); P3
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 5 vertices
```

```sage
P4 = Polyhedron(vertices=[(-1,1,0),(1,1,0),(-1,0,1), (1,0,1),(0,-1,1),(0,1,1)]); P4
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 6 vertices
```

```sage
P3.plot() + P4.plot()
Graphics3d Object
```

```sage
(P3+P4).plot()
Graphics3d Object
```

We can also find the intersection of two polytopes... and this too has two syntaxes!

```sage
int12 = P1.intersection(P2*.5); int12.plot()
Graphics object consisting of 7 graphics primitives
```

```sage
int34 = P3 & P4; int34.plot()
Graphics3d Object
```

Should one wish to translate, one can.

```sage
transP2 = P2.translation([2,1])
sage: P2.plot() + transP2.plot()
Graphics object consisting of 14 graphics primitives
```

Then of course we can take prisms, pyramids, and bipyramids of polytopes...

```sage
P2.prism().plot()
Graphics3d Object
```

```sage
P1.pyramid().plot()
Graphics3d Object
```

```sage
P2dual.bipyramid().plot()
Graphics3d Object
```

Okay, fine. Yes, Sage has some kinds of polytopes built in. If you type polytopes. and then press TAB after the period, you’ll get a list of pre-built polytopes.
Let's look at a 4-dimensional polytope.

```
sage: P8 = polytopes.hypercube(4)
sage: P8.plot()
Graphics3d Object
```

We can see it from a different perspective:

```
sage: P8.schlegel_projection(position=1/2).plot()
Graphics3d Object
```

**Queries to polytopes**

Once you've constructed some polytope, you can ask Sage questions about it.

```
sage: P1.contains([1,0])
True
```

```
sage: P1.interior_contains([3,0])
False
```

```
sage: P3.contains([1,0,0])
False
```

Face information can be useful.

```
sage: int34.f_vector()
(1, 8, 12, 6, 1)
```

Well, geometric information might be more helpful... Here we are told which of the vertices form each 2-face:

```
sage: [f.ambient_V_indices() for f in int34.faces(2)]
[(2, 6, 7), (0, 1, 3, 5), (1, 3, 4), (0, 5, 6, 7), (0, 1, 2, 4, 6), (2, 3, 4, 5, 7)]
```

Yeah, that isn't so useful as it is. Let's figure out the vertex and hyperplane representations of the first face in the list.

```
sage: first2faceofint34 = int34.faces(2)[0]
sage: first2faceofint34.ambient_Hrepresentation(); first2faceofint34.vertices()
(An inequality (0, 0, -1) x + 1 >= 0,)
(A vertex at (2/3, 2/15, 1), A vertex at (3/8, 1/10, 1), A vertex at (1/2, 3/10, 1))
```

If you want more... Base class for polyhedra: Miscellaneous methods is the first place you want to go.
A Longer Introduction to Polyhedral Computations in Sage

Author: Jean-Philippe Labbé

This tutorial aims to showcase some of the possibilities of Sage concerning polyhedral geometry and combinatorics.

The classic literature on the topic includes:
- Convex Polytopes, Branko Grünbaum, [Gru1967]
- An Introduction to Convex Polytopes, Arne Brøndsted, [Bro1983]
- Lectures on Polytopes, Günter M. Ziegler, [Zie2007]

For the nomenclature, Sage tries to follow:
- Handbook of Discrete and Computational Geometry, Chapter 15, [Goo2004]

Preparation of this document was supported in part by the OpenDreamKit project and written during the SageDays 84 in Olot (Spain).

Lectures Menu

- A Longer Introduction to Polyhedral Computations in Sage
  - Lecture 0: Basic definitions and constructions
  - Lecture 1: Representation objects
  - Lecture 2: Backends for polyhedral computations
  - Lecture 3: To every polyhedron, the proper parent class
  - Lecture 4: A library of polytopes
  - Bibliography

Lecture 0: Basic definitions and constructions

A real \((k \times d)\)-matrix \(A\) and a real vector \(b\) in \(\mathbb{R}^d\) define a (convex) polyhedron \(P\) as the set of solutions of the system of linear inequalities:

\[
A \cdot x + b \geq 0.
\]

Each row of \(A\) defines a closed half-space of \(\mathbb{R}^d\). Hence a polyhedron is the intersection of finitely many closed half-spaces in \(\mathbb{R}^d\). The matrix \(A\) may contain equal rows, which may lead to a set of equalities satisfied by the polyhedron. If there are no redundant rows in the above definition, this definition is referred to as the \(H\)-representation of a polyhedron.

A maximal affine subspace \(L\) contained in a polyhedron is a lineality space of \(P\). Fixing a point \(o\) of the lineality space \(L\) to act as the origin, one can write every point \(p\) inside a polyhedron as a combination

\[
p = \ell + \sum_{i=1}^{n} \lambda_i v_i + \sum_{i=1}^{m} \mu_i r_i,
\]

where \(\ell \in L\) (using \(o\) as the origin), \(\sum_{i=1}^{n} \lambda_i = 1\), \(\lambda_i \geq 0\), \(\mu_i \geq 0\), and \(r_i \neq 0\) for all \(0 \leq i \leq m\) and the set of \(r_i\)'s are positively independent (the origin is not in their positive span). For a given point \(p\) there may be many equivalent ways to write the above using different sets \(\{v_i\}_{i=1}^{n}\) and \(\{r_i\}_{i=1}^{m}\). Hence we require the sets to be inclusion minimal sets such that we can get the above property equality for any point \(p \in P\).
The vectors $v_i$ are called the vertices of $P$ and the vectors $r_i$ are called the rays of $P$. This way to represent a polyhedron is referred to as the $V$-representation of a polyhedron. The first sum represents the convex hull of the vertices $v_i$'s and the second sum represents a pointed polyhedral cone generated by finitely many rays.

When the lineality space and the rays are reduced to a point (i.e. no rays and no lines) the object is often referred to as a polytope.

**Note:** As mentioned in the documentation of the constructor when typing `Polyhedron?`:

You may either define it with vertex/ray/line or inequalities/equations data, but not both. Redundant data will automatically be removed (unless “minimize=False”), and the complementary representation will be computed.

Here is the documentation for the constructor function of Polyhedra.

### $V$-representation

First, let’s define a polyhedron object as the convex hull of a set of points and some rays.

```python
sage: P1 = Polyhedron(vertices = [[1, 0], [0, 1]], rays = [[1, 1]])
sage: P1
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 2 vertices and 1 ray
```

The string representation already gives a lot of information:

- the dimension of the polyhedron (the smallest affine space containing it)
- the dimension of the space in which it is defined
- the base ring ($\mathbb{Z}^2$) over which the polyhedron lives (this specifies the parent class, see Parents for Polyhedra)
- the number of vertices
- the number of rays

Of course, you want to know what this object looks like:

```python
sage: P1.plot()
Graphics object consisting of 5 graphics primitives
```

We can also add a lineality space.

```python
sage: P2 = Polyhedron(vertices = [[1/2, 0, 0], [0, 1/2, 0]],
       ....: rays = [[1, 1, 0]],
       ....: lines = [[0, 0, 1]])
sage: P2
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 2 vertices, 1 ray, 1 line
sage: P2.plot()
Graphics3d Object
```

Notice that the base ring changes because of the value $\frac{1}{2}$. Indeed, Sage finds an appropriate ring to define the object.

```python
sage: P1.parent()
Polyhedra in QQ^2
sage: P2.parent()
Polyhedra in QQ^3
```
The chosen ring depends on the input format.

```
sage: P3 = Polyhedron(vertices = [[0.5, 0], [0, 0.5]])
sage: P3
A 1-dimensional polyhedron in RDF^2 defined as the convex hull of 2 vertices
sage: P3.parent()
Polyhedra in RDF^2
```

**Warning:** The base ring RDF should be used with care. As it is not an exact ring, certain computations may break or silently produce wrong results, for example when dealing with non-simplicial polyhedra.

The following example demonstrates the limitations of RDF.

```
sage: D = polytopes.dodecahedron()                      # optional -
                          # sage.rings.number_field
sage: D                                                      # optional -
                          # sage.rings.number_field
A 3-dimensional polyhedron
 in (Number Field in sqrt5 with defining polynomial x^2 - 5
  with sqrt5 = 2.236067774997990?)^3
 defined as the convex hull of 20 vertices
sage: vertices_RDF = [n(v.vector(),digits=6) for v in D.vertices()]  # optional -
                          # sage.rings.number_field
sage: D_RDF = Polyhedron(vertices=vertices_RDF, base_ring=RDF)    # optional -
                          # sage.rings.number_field
```

doctest:warning...
UserWarning: This polyhedron data is numerically complicated; cdd could not convert between the inexact V and H representation without loss of data. The resulting object might show inconsistencies.

```
sage: D_RDF = Polyhedron(vertices=sorted(vertices_RDF), base_ring=RDF) # optional -
                          # sage.rings.number_field
```

```
...<br>
ValueError: *Error: Numerical inconsistency is found. Use the GMP exact arithmetic.
```

If the input of the polyhedron consists of python float, it automatically converts the data to RDF:

```
sage: Polyhedron(vertices=[[float(1.1)]])
A 0-dimensional polyhedron in RDF^1 defined as the convex hull of 1 vertex
```

It is also possible to define a polyhedron over algebraic numbers.

```
sage: sqrt_2 = AA(2)^(1/2)                              # optional -
                          # sage.rings.number_field
sage: cbrt_2 = AA(2)^(1/3)                               # optional -
                          # sage.rings.number_field
sage: timeit('Polyhedron(contacts=[sqrt_2, 0, [sqrt_2, cbart_2]])')  # optional -
                          # sage.rings.number_field # random
```

(continues on next page)

12.1. Thematic tutorial document tree 319
sage: P4 = Polyhedron(vertices = [[sqrt_2, 0], [0, cbrt_2]]); P4  # optional -
˓→sage.rings.number_field
A 1-dimensional polyhedron in AA^2 defined as the convex hull of 2 vertices

There is another way to create a polyhedron over algebraic numbers:

sage: K.<a> = NumberField(x^2 - 2, embedding=AA(2)**(1/2))  # optional -
˓→sage.rings.number_field
sage: L.<b> = NumberField(x^3 - 2, embedding=AA(2)**(1/3))  # optional -
˓→sage.rings.number_field
sage: timeit('Polyhedron(vertices = [[a, 0], [0, b]])')  # optional -
˓→sage.rings.number_field # random
5 loops, best of 3: 39.9 ms per loop
sage: P5 = Polyhedron(vertices = [[a, 0], [0, b]]); P5  # optional -
˓→sage.rings.number_field
A 1-dimensional polyhedron in AA^2 defined as the convex hull of 2 vertices

If the base ring is known it may be a good option to use the proper `sage.rings.number_field.number_field`. number_field.composite_fields():

sage: J = K.composite_fields(L)[0]  # optional -
˓→sage.rings.number_field
sage: timeit('Polyhedron(vertices = [[J(a), 0], [0, J(b)]])')  # optional -
˓→sage.rings.number_field # random
25 loops, best of 3: 9.8 ms per loop
sage: P5_comp = Polyhedron(vertices = [[J(a), 0], [0, J(b)]]); P5_comp  # optional -
˓→sage.rings.number_field
A 1-dimensional polyhedron in (Number Field in ab with defining polynomial
x^6 - 6*x^4 - 4*x^3 + 12*x^2 - 24*x - 4
with ab = -0.1542925124782219?)^2
defined as the convex hull of 2 vertices

Polyhedral computations with the Symbolic Ring are not implemented. It is not possible to define a polyhedron over it:

sage: sqrt_2s = sqrt(2)  # optional -
˓→sage.symbolic
sage: cbrt_2s = 2**(1/3)  # optional -
˓→sage.symbolic
sage: Polyhedron(vertices = [[sqrt_2s, 0], [0, cbrt_2s]])  # optional -
˓→sage.symbolic
Traceback (most recent call last):
...
ValueError: no default backend for computations with Symbolic Ring

Similarly, it is not possible to create polyhedron objects over RR (no matter how many bits of precision).

sage: F45 = RealField(45)
sage: F100 = RealField(100)
sage: f = 1.1
sage: Polyhedron(vertices=[[F45(f)])
Traceback (most recent call last):
Thematic Tutorials, Release 9.7

ValueError: the only allowed inexact ring is 'RDF' with backend 'cdd'
sage: Polyhedron(vertices=[[F100(f)]]))
Traceback (most recent call last):
...
ValueError: the only allowed inexact ring is 'RDF' with backend 'cdd'

There is one exception, when the number of bits of precision is 53, then the base ring is converted to RDF:

sage: F53 = RealField(53)
sage: Polyhedron(vertices=[[F53(f)]]))
A 0-dimensional polyhedron in RDF^1 defined as the convex hull of 1 vertex
sage: type(Polyhedron(vertices=[[F53(f)]]))
<class 'sage.geometry.polyhedron.parent.Polyhedra_RDF_cdd_with_category.element_class'>

This behavior can be seen as wrong, but it allows the following to be acceptable by Sage:

sage: Polyhedron([[1.0, 2.3], (3.5, 2.0)])
A 1-dimensional polyhedron in RDF^2 defined as the convex hull of 2 vertices

without having specified the base ring RDF by the user.

**H-representation**

If a polyhedron object was constructed via a V-representation, Sage can provide the H-representation of the object.

sage: for h in P1.Hrepresentation():
....:    print(h)
An inequality (1, 1) x - 1 >= 0
An inequality (1, -1) x + 1 >= 0
An inequality (-1, 1) x + 1 >= 0

Each line gives a row of the matrix A and an entry of the vector b. The variable x is a vector in the ambient space where P1 is defined. The H-representation may contain equations:

sage: P3.Hrepresentation()
(An inequality (-2.0, 0.0) x + 1.0 >= 0,
An inequality (1.0, 0.0) x + 0.0 >= 0,
An equation (1.0, 1.0) x - 0.5 == 0)

The construction of a polyhedron object via its H-representation, requires a precise format. Each inequality \((a_1, \ldots, a_id) \cdot x + b_i \geq 0\) must be written as \([b_i, a_i1, \ldots, a_id]\).

sage: P3_H = Polyhedron(ieqs = [[1.0, -2, 0], [0, 1, 0]], eqns = [[-0.5, 1, 1]])
sage: P3 == P3_H
True
sage: P3_H.Vrepresentation()
(A vertex at (0.0, 0.5), A vertex at (0.5, 0.0))

It is worth using the parameter eqns to shorten the construction of the object. In the following example, the first four rows are the negative of the second group of four rows.
sage: H = [[0, 0, 0, 0, 0, 0, 0, 1],
....:       [0, 0, 0, 0, 0, 0, 1, 0],
....:       [-2, 1, 1, 1, 1, 0, 0, 0],
....:       [0, 0, 0, 0, 0, 0, 0, 1],
....:       [0, 0, 0, 0, 0, 0, -1, 0],
....:       [2, -1, -1, -1, -1, -1, 0, 0],
....:       [0, 0, 0, 0, 0, 0, 0, -1],
....:       [2, -1, -1, -1, -1, 0, 0, 0],
....:       [0, 0, 0, 0, 0, 0, 0, 0],
....:       [0, 0, 0, 0, 0, 0, 0, 0],
....:       [0, 0, 1, 0, 0, 0, 0, 0],
....:       [0, 0, 1, 0, 0, 0, 0, 0],
....:       [0, 0, -1, 0, 0, 0, 0, 0],
....:       [1, 0, 0, 0, -1, 0, 0, 0],
....:       [1, 0, 0, 0, 0, 0, 0, 0],
....:       [1, 0, 0, 0, 0, 0, 0, 0],
....:       [1, -1, 0, 0, 0, 0, 0, 0]]

sage: timeit('Polyhedron(ieqs = H)')  # random
125 loops, best of 3: 5.99 ms per loop
sage: timeit('Polyhedron(ieqs = H[8:], eqns = H[:4])')  # random
125 loops, best of 3: 4.78 ms per loop
sage: Polyhedron(ieqs = H) == Polyhedron(ieqs = H[8:], eqns = H[:4])
True

Of course, this is a toy example, but it is generally worth to preprocess the data before defining the polyhedron if possible.

Lecture 1: Representation objects

Many objects are related to the $H$- and $V$-representations. Sage has classes implemented for them.

$H$-representation

You can store the $H$-representation in a variable and use the inequalities and equalities as objects.

sage: P3_QQ = Polyhedron(vertices = [[0.5, 0], [0, 0.5]], base_ring=QQ)
sage: HRep = P3_QQ.Hrepresentation()
sage: H1 = HRep[0]; H1
An equation (2, 2) x - 1 == 0
sage: H2 = HRep[1]; H2
An inequality (0, -2) x + 1 >= 0
sage: H1.<tab>  # not tested
sage: H1.A()
(2, 2)
sage: H1.b()
-1
sage: H1.is_equation()
True
sage: H1.is_inequality()
False
sage: H1.contains(vector([0,0]))
False
sage: H2.contains(vector([0,0]))
True
sage: H1.is_incident(H2)
True

It is possible to obtain the different objects of the $H$-representation as follows.

```python
sage: P3_QQ.equations()
(An equation (2, 2) x - 1 == 0,)
sage: P3_QQ.inequalities()
(An inequality (0, -2) x + 1 >= 0, An inequality (0, 1) x + 0 >= 0)
```

**Note:** It is recommended to use `equations` or `equation_generator` (and similarly for inequalities) if one wants to iterate over them instead of `equations_list`.

---

**$V$-representation**

Similarly, you can access vertices, rays and lines of the polyhedron.

```python
sage: VRep = P2.Vrepresentation(); VRep
(A line in the direction (0, 0, 1),
 A vertex at (0, 1/2, 0),
 A vertex at (1/2, 0, 0),
 A ray in the direction (1, 1, 0))
sage: L = VRep[0]; L
A line in the direction (0, 0, 1)
sage: V = VRep[1]; V
A vertex at (0, 1/2, 0)
sage: R = VRep[3]; R
A ray in the direction (1, 1, 0)
sage: L.is_line()
True
sage: L.is_incident(V)
True
sage: R.is_incident(V)
False
sage: L.vector()
(0, 0, 1)
sage: V.vector()
(0, 1/2, 0)
```

It is possible to obtain the different objects of the $V$-representation as follows.

```python
sage: P2.vertices()
(A vertex at (0, 1/2, 0), A vertex at (1/2, 0, 0))
sage: P2.rays()
(A ray in the direction (1, 1, 0),)
sage: P2.lines()
```

(continues on next page)
(A line in the direction (0, 0, 1),)

\[
\begin{bmatrix}
0 & 1/2 \\
1/2 & 0 \\
0 & 0
\end{bmatrix}
\]

**Note:** It is recommended to use `vertices` or `vertex_generator` (and similarly for rays and lines) if one wants to iterate over them instead of `vertices_list`.

---

**Lecture 2: Backends for polyhedral computations**

To deal with polyhedron objects, Sage currently has four backends available. These backends offer various functionalities and have their own specific strengths and limitations.

- **The cdd backend for polyhedral computations**
  - The cdd and cddplus homepage
- **The PPL (Parma Polyhedra Library) backend for polyhedral computations**
  - The Parma Polyhedra Library homepage
- **The polymake backend for polyhedral computations**
  - The polymake project for convex geometry
- **The Python backend**
  - This is a **python** backend that provides an implementation of polyhedron over irrational coordinates.
- **The Normaliz backend for polyhedral computations**, (requires the optional package `pynormaliz`)
  - Normaliz Homepage

The default backend is `ppl`. Whenever one needs **speed** it is good to try out the different backends. The backend `field` is **not** specifically designed for dealing with extremal computations but can do computations in exact coordinates.

**The cdd backend**

In order to use a specific backend, we specify the `backend` parameter.

```python
sage: P1_cdd = Polyhedron(verticestes = [[1, 0], [0, 1]], rays = [[1, 1]], backend='cdd')
```

```
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 2 vertices and 1 ray
```

A priori, it seems that nothing changed, but...

```python
sage: P1_cdd.parent()
Polyhedra in QQ^2
```

The polyhedron `P1_cdd` is now considered as a rational polyhedron by the backend `cdd`. We can also check the backend and the parent using `type`:
We see
  • the backend used (ex: backend_cdd)
  • followed by a dot ‘.’
  • the parent (ex: Polyhedra_QQ) followed again by the backend,
and you can safely ignore the rest for the purpose of this tutorial.

The cdd backend accepts also entries in RDF:

```sage
sage: P3_cdd = Polyhedron(vertices = [[0.5, 0], [0, 0.5]], backend='cdd')
sage: P3_cdd
A 1-dimensional polyhedron in RDF^2 defined as the convex hull of 2 vertices
```

but not algebraic or symbolic values:

```sage
sage: P4_cdd = Polyhedron(vertices = [[sqrt_2, 0], [0, cbrt_2]], backend='cdd')  # optional - sage.rings.number_field
Traceback (most recent call last):
... ValueError: No such backend (=cdd) implemented for given basering (=Algebraic Real Field).
```

```sage
sage: P5_cdd = Polyhedron(vertices = [[sqrt_2s, 0], [0, cbrt_2s]], backend='cdd')  # optional - sage.symbolic
Traceback (most recent call last):
... ValueError: No such backend (=cdd) implemented for given basering (=Symbolic Ring).
```

It is possible to get the cdd format of any polyhedron object defined over \( \mathbb{Z}, \mathbb{Q}, \) or RDF:

```sage
sage: print(P1.cdd_Vrepresentation())
V-representation
begin
  3 3 rational
  0 1 1
  1 0 1
  1 1 0
end
```

```sage
sage: print(P3.cdd_Hrepresentation())
H-representation
linearity 1 1
begin
  3 3 real
-0.5 1.0 1.0
1.0 -2.0 0.0
0.0 1.0 0.0
end
```
You can also write this data to a file using the method `write_cdd_Hrepresentation(filename)` or `write_cdd_Vrepresentation(filename)`, where `filename` is a string containing a path to a file to be written.

**The ppl backend**

The default backend for polyhedron objects is `ppl`.

```python
sage: type(P1)
<class 'sage.geometry.polyhedron.parent.Polyhedra_QQ_ppl_with_category.element_class'>
sage: type(P2)
<class 'sage.geometry.polyhedron.parent.Polyhedra_QQ_ppl_with_category.element_class'>
sage: type(P3)  # has entries like 0.5
<class 'sage.geometry.polyhedron.parent.Polyhedra_RDF_cdd_with_category.element_class'>
```

As you see, it does not accepts values in `RDF` and the polyhedron constructor used the `cdd` backend.

**The polymake backend**

The `polymake` backend is provided when the experimental package polymake for sage is installed.

```python
sage: p = Polyhedron(vertices=[(0,0),(1,0),(0,1)],
                    ....: rays=[(1,1)], lines=[],
                    ....: backend='polymake', base_ring=QQ)
```

An example with quadratic field:

```python
sage: V = polytopes.dodecahedron().vertices_list()  # optional - sage.rings.number_field
sage: Polyhedron(vertices=V, backend='polymake')  # optional - jupymake
```

A 3-dimensional polyhedron
in (Number Field in sqrt5 with defining polynomial x^2 - 5
with sqrt5 = 2.236067977499790?)^3
defined as the convex hull of 20 vertices

**The field backend**

As it turns out, the rational numbers do not suffice to represent all combinatorial types of polytopes. For example, Perles constructed a 8-dimensional polytope with 12 vertices which does not have a realization with rational coordinates, see Example 6.21 p. 172 of [Zie2007]. Furthermore, if one wants a realization to have specific geometric property, such as symmetry, one also sometimes need irrational coordinates.

The backend `field` provides the necessary tools to deal with such examples.

```python
sage: type(D)  # optional - sage.rings.number_field
<class 'sage.geometry.polyhedron.parent.Polyhedra_field_with_category.element_class'>
```

Any time that the coordinates should be in an extension of the rationals, the backend `field` is called.
The normaliz backend

The fourth backend is normaliz and is an optional Sage package.

```
sage: P1_normaliz = Polyhedron(vertices = [[1, 0], [0, 1]], rays = [[1, 1]], backend='normaliz')  # optional - pynormaliz
sage: type(P1_normaliz)  # optional - pynormaliz
<class 'sage.geometry.polyhedron.parent.Polyhedra_QQ_normaliz_with_category.element_class'>
```

This backend does not work with RDF or other inexact fields.

```
sage: P3_normaliz = Polyhedron(vertices = [[0.5, 0], [0, 0.5]], backend='normaliz')  # optional - pynormaliz
Traceback (most recent call last):
  ... ValueError: No such backend (=normaliz) implemented for given basering (=Real DoubleField).
```

The normaliz backend provides fast computations with algebraic numbers. They can be entered as elements of an embedded number field, the field of algebraic numbers, or even the symbolic ring. Internally the computation is done using an embedded number field.

```
sage: P4_normaliz = Polyhedron(vertices = [[sqrt_2, 0], [0, cbrt_2]], backend='normaliz')  # optional - pynormaliz
sage: P5_normaliz = Polyhedron(vertices = [[sqrt_2s, 0], [0, cbrt_2s]], backend='normaliz')  # optional - pynormaliz
```

... (continues on next page)
Thematic Tutorials, Release 9.7

Continued from previous page:

```
sage: P5_normaliz
˓→ # optional - pynormaliz
A 1-dimensional polyhedron in (Symbolic Ring)^2 defined as the convex hull of 2 vertices
```

The backend `normaliz` provides other methods such as `integral_hull`, which also works on unbounded polyhedron.

```
sage: P6 = Polyhedron(vertices = [[0, 0], [3/2, 0], [3/2, 3/2], [0, 3]], backend='normaliz')  # optional - pynormaliz
sage: IH = P6.integral_hull(); IH  # optional - pynormaliz
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 4 vertices
sage: P6.plot(color='blue')+IH.plot(color='red')  # optional - pynormaliz
˓→ # optional - pynormaliz
```

Graphics object consisting of 12 graphics primitives

```
sage: P1_normaliz.integral_hull()  # optional - pynormaliz
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 2 vertices and 1 ray
```

Lecture 3: To every polyhedron, the proper parent class

In order to **know all the methods that a polyhedron object has** one has to look into its **base class**:

- Base class for polyhedra: Miscellaneous methods: This is the generic class for Polyhedron related objects.
- Base class for polyhedra over Z
- Base class for polyhedra over Q
- Base class for polyhedra over RDF

Don’t be surprised if the classes look empty! The classes mainly contain private methods that implement some comparison methods: to verify equality and inequality of numbers in the base ring and other internal functionalities.

To get a full overview of methods offered to you, **Base class for polyhedra: Miscellaneous methods** is the first place you want to go.

Lecture 4: A library of polytopes

There are a lot of polytopes that are readily available in the library, see **Library of commonly used, famous, or interesting polytopes**. Have a look at them to see if your polytope is already defined!

```
sage: A = polytopes.buckyball(); A  # can take long  # optional - sage.rings.number_field
˓→ sage.rings.number_field
A 3-dimensional polyhedron in (Number Field in sqrt5 with defining polynomial x^2 - 5 with sqrt5 = 2.236067977499790?)^3 defined as the convex hull of 60 vertices
sage: B = polytopes.cross_polytope(4); B
A 4-dimensional polyhedron in ZZ^4 defined as the convex hull of 8 vertices
sage: C = polytopes.cyclic_polytope(3,10); C
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 10 vertices
sage: E = polytopes.snub_cube(exact=False); E
```

(continues on next page)
A 3-dimensional polyhedron in RDF^3 defined as the convex hull of 24 vertices

```
sage: polytopes.<tab> # not tested, to view all the possible polytopes
```

### Bibliography

Quick reference for polyhedra in Sage

Author: Jean-Philippe Labbé <labbe@math.fu-berlin.de> Vincent Delecroix <vincent.delecroix@u-bordeaux.fr>

### List of Polyhedron methods

#### H and V-representation

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>base_ring()</code></td>
<td>ring on which the polyhedron is defined</td>
</tr>
<tr>
<td><code>ambient_space()</code></td>
<td>ambient vector space or free module</td>
</tr>
<tr>
<td><code>Hrepresentation_space()</code></td>
<td>vector space or free module used for the vectors of the H-representation</td>
</tr>
<tr>
<td><code>Vrepresentation_space()</code></td>
<td>vector space or free module used for the vectors of the V-representation</td>
</tr>
<tr>
<td><code>n_Hrepresentation()</code></td>
<td>number of elements in the H-representation (sum of the number of equations and inequalities)</td>
</tr>
<tr>
<td><code>n_Vrepresentation()</code></td>
<td>number of elements in the V-representation (sum of vertices, rays and lines)</td>
</tr>
<tr>
<td><code>n_equations()</code></td>
<td>number of equations</td>
</tr>
<tr>
<td><code>n_inequalities()</code></td>
<td>number of inequalities</td>
</tr>
<tr>
<td><code>n_vertices()</code></td>
<td>number of vertices</td>
</tr>
<tr>
<td><code>n_rays()</code></td>
<td>number of rays</td>
</tr>
<tr>
<td><code>n_lines()</code></td>
<td>number of lines</td>
</tr>
<tr>
<td><code>n_facets()</code></td>
<td>number of facets</td>
</tr>
</tbody>
</table>

### Polyhedron boolean properties:


<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>is_empty()</code></td>
<td>tests emptiness</td>
</tr>
<tr>
<td><code>is_universe()</code></td>
<td>tests whether a polyhedra is the whole ambient space</td>
</tr>
<tr>
<td><code>is_full_dimensional()</code></td>
<td>tests if the polyhedron has the same dimension as the ambient space</td>
</tr>
<tr>
<td><code>is_combinatorially_isomorphic()</code></td>
<td>whether two polyhedra are combinatorially isomorphic</td>
</tr>
<tr>
<td><code>is_compact()</code></td>
<td>tests compactness, or boundedness of a polyhedron</td>
</tr>
<tr>
<td><code>is_lattice_polytope()</code></td>
<td>tests whether a polyhedron is a lattice polytope</td>
</tr>
<tr>
<td><code>is_inscribed()</code></td>
<td>tests whether the polyhedron is inscribed in a sphere</td>
</tr>
<tr>
<td><code>is_minkowski_summand()</code></td>
<td>tests if the polyhedron can be used to produce another given polyhedron using a Minkowski sum.</td>
</tr>
<tr>
<td><code>is_neighborly()</code></td>
<td>tests whether the polyhedron has full skeleton until half of the dimension (or up to a certain dimension)</td>
</tr>
<tr>
<td><code>is_reflexive()</code></td>
<td>tests if the polar of a lattice polytope is also a lattice polytope (only for Polyhedron over ZZ)</td>
</tr>
<tr>
<td><code>is_simple()</code></td>
<td>checks whether the degree of all vertices is equal to the dimension of the polytope</td>
</tr>
<tr>
<td><code>is_simplex()</code></td>
<td>test whether a polytope is a simplex</td>
</tr>
<tr>
<td><code>is_simplicial()</code></td>
<td>checks whether all faces of the polyhedron are simplices</td>
</tr>
<tr>
<td><code>is_lawrence_polytope()</code></td>
<td>tests whether self is a Lawrence polytope</td>
</tr>
<tr>
<td><code>is_self_dual()</code></td>
<td>tests whether the polytope is self-dual</td>
</tr>
<tr>
<td><code>is_pyramid()</code></td>
<td>test whether the polytope is a pyramid over one of its facets</td>
</tr>
<tr>
<td><code>is_bipyramid()</code></td>
<td>test whether the polytope is combinatorially equivalent to a bipyramid over some polytope</td>
</tr>
<tr>
<td><code>is_prism()</code></td>
<td>test whether the polytope is combinatorially equivalent to a prism of some polytope</td>
</tr>
</tbody>
</table>

### Enumerative properties

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ambient_dim()</code></td>
<td>the dimension of the ambient vector space</td>
</tr>
<tr>
<td><code>dim()</code></td>
<td>the dimension of the polytope</td>
</tr>
<tr>
<td><code>dimension()</code></td>
<td>alias of dim</td>
</tr>
<tr>
<td><code>f_vector()</code></td>
<td>the ( f )-vector (number of faces of each dimension)</td>
</tr>
<tr>
<td><code>flag_f_vector()</code></td>
<td>the flag-( f )-vector (number of chains of faces)</td>
</tr>
<tr>
<td><code>neighborliness()</code></td>
<td>highest cardinality for which all ( k )-subsets of the vertices are faces of the polytope</td>
</tr>
<tr>
<td><code>simpliciality()</code></td>
<td>highest cardinality for which all ( k )-faces are simplices</td>
</tr>
<tr>
<td><code>simplicity()</code></td>
<td>highest cardinality for which the polar is ( k )-simplicial</td>
</tr>
</tbody>
</table>

### Implementation properties

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>backend()</code></td>
<td>gives the backend used</td>
</tr>
<tr>
<td><code>base_ring()</code></td>
<td>gives the base ring used</td>
</tr>
<tr>
<td><code>change_ring()</code></td>
<td>changes the base ring</td>
</tr>
</tbody>
</table>

### Transforming polyhedra
### Minkowski Sum

- **minkowski_sum()**
  - Minkowski sum of two polyhedra

### Minkowski Difference

- **minkowski_difference()**
  - Minkowski difference of two polyhedra

### Minkowski Decompositions

- **minkowski_decompositions()**
  - Minkowski decomposition (only for `Polyhedron over ZZ`)

### Cartesian Product

- **product()**
  - Cartesian product of two polyhedra

### Intersection

- **intersection()**
  - Intersection of two polyhedra

### Join

- **join()**
  - Join of two polyhedra

### Convex Hull

- **convex_hull()**
  - Convex hull of the union of two polyhedra

### Barycentric Subdivision

- **barycentric_subdivision()**
  - Constructs a geometric realization of the barycentric subdivision

### Dilation

- **dilation()**
  - Scalar dilation

### Face Truncation

- **face_truncation()**
  - Truncates a specific face

### Face Split

- **face_split()**
  - Returns the face splitting of a face of self

### One Point Suspension

- **one_point_suspension()**
  - The one-point suspension over a vertex of self (face splitting of a vertex)

### Stack

- **stack()**
  - Stack a face of the polyhedron

### Lattice Polytope

- **lattice_polytope()**
  - Returns an encompassing lattice polytope.

### Polar

- **polar()**
  - Returns the polar of a polytope (needs to be compact)

### Prism

- **prism()**
  - Prism over a polyhedron (increases both the dimension of the polyhedron and the dimension of the ambient space)

### Pyramid

- **pyramid()**
  - Pyramid over a polyhedron (increases both the dimension of the polyhedron and the dimension of the ambient space)

### Bipyramid

- **bipyramid()**
  - Bipyramid over a polyhedron (increases both the dimension of the polyhedron and the dimension of the ambient space)

### Translation

- **translation()**
  - Translates by a given vector

### Truncation

- **truncation()**
  - Truncates all vertices simultaneously

### Lawrence Extension

- **lawrence_extension()**
  - Returns the Lawrence extension of self on a given point

### Lawrence Polytope

- **lawrence_polytope()**
  - Returns the Lawrence polytope of self

### Wedge

- **wedge()**
  - Returns the wedge over a face of self

### Combinatorics

- **combinatorial_polyhedron()**
  - The combinatorial polyhedron

- **face_lattice()**
  - The face lattice

- **hasse_diagram()**
  - The Hasse diagram

- **combinatorial_automorphism_group()**
  - Automorphism group of the underlying combinatorial polytope

- **graph(), vertex_graph()**
  - Underlying graph

- **vertex_digraph()**
  - Digraph (orientation of edges determined by a linear form)

- **vertex_facet_graph()**
  - Bipartite digraph given vertex-facet adjacency

- **adjacency_matrix()**
  - Adjacency matrix

- **incidence_matrix()**
  - Incidence matrix

- **slack_matrix()**
  - Slack matrix

- **facet_adjacency_matrix()**
  - Adjacency matrix of the facets

- **vertex_adjacency_matrix()**
  - Adjacency matrix of the vertices

### Integral Points

---

12.1. Thematic tutorial document tree

331
### Thematic Tutorials, Release 9.7

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ehrhart_polynomial()</td>
<td>the Ehrhart polynomial for ( \text{Polyhedron over ZZ} )</td>
</tr>
<tr>
<td>ehrhart_quasipolynomial()</td>
<td>the Ehrhart quasipolynomial for ( \text{Polyhedron over QQ} )</td>
</tr>
<tr>
<td>h_star_vector()</td>
<td>the ( h ) -vector for polytopes with integral vertices</td>
</tr>
<tr>
<td>integral_points()</td>
<td>list of integral points</td>
</tr>
<tr>
<td>integral_points_count()</td>
<td>number of integral points</td>
</tr>
<tr>
<td>get_integral_point()</td>
<td>get the ( i )-th integral point without computing all interior lattice points</td>
</tr>
<tr>
<td>has_IP_property()</td>
<td>checks whether the origin is an interior lattice point and compactness (only for ( \text{Polyhedron over ZZ} ))</td>
</tr>
<tr>
<td>random_integral_point()</td>
<td>get a random integral point</td>
</tr>
</tbody>
</table>

### Getting related geometric objects

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>affine_hull()</td>
<td>returns the smallest affine subspace containing the polyhedron</td>
</tr>
<tr>
<td>boundary_complex()</td>
<td>returns the boundary complex of simplicial compact polyhedron</td>
</tr>
<tr>
<td>center()</td>
<td>returns the average of the vertices of the polyhedron</td>
</tr>
<tr>
<td>centroid()</td>
<td>returns the center of the mass</td>
</tr>
<tr>
<td>representative_point()</td>
<td>returns the sum of the center and the rays</td>
</tr>
<tr>
<td>a_maximal_chain()</td>
<td>returns a maximal chain of faces</td>
</tr>
<tr>
<td>face_fan()</td>
<td>returns the fan spanned by the faces of the polyhedron</td>
</tr>
<tr>
<td>face_generator()</td>
<td>a generator over the faces</td>
</tr>
<tr>
<td>faces()</td>
<td>the list of faces</td>
</tr>
<tr>
<td>facets()</td>
<td>the list of facets</td>
</tr>
<tr>
<td>join_of_Vrep()</td>
<td>returns the smallest face containing specified Vrepresentatives</td>
</tr>
<tr>
<td>least_common_superface_of_Vrep()</td>
<td>returns the largest face contained in specified Vrepresentatives</td>
</tr>
<tr>
<td>meet_of_Hrep()</td>
<td>returns the fan spanned by the normals of the supporting hyperplanes of the polyhedron</td>
</tr>
<tr>
<td>greatest_common_subface_of_Hrep()</td>
<td>returns the (affine) Gale transform of the vertices of the polyhedron</td>
</tr>
<tr>
<td>normal_fan()</td>
<td>returns the hyperplane arrangement given by the defining facets of the polyhedron</td>
</tr>
<tr>
<td>hyperplane_arrangement()</td>
<td>transform the polyhedra into a Linear Program</td>
</tr>
<tr>
<td>to_linear_program()</td>
<td>returns a triangulation of the polyhedron</td>
</tr>
<tr>
<td>triangulate()</td>
<td>returns an iterator of the fibrations of the lattice polytope (only for ( \text{Polyhedron over ZZ} ))</td>
</tr>
</tbody>
</table>

### Other
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bounded_edges()</td>
<td>generator for bounded edges</td>
</tr>
<tr>
<td>bounding_box()</td>
<td>returns the vertices of an encompassing cube</td>
</tr>
<tr>
<td>contains()</td>
<td>tests whether the polyhedron contains a vector</td>
</tr>
<tr>
<td>interior_contains()</td>
<td>tests whether the polyhedron contains a vector in its interior using the ambient topology</td>
</tr>
<tr>
<td>relative_interior_contains()</td>
<td>tests whether the polyhedron contains a vector in its relative interior</td>
</tr>
<tr>
<td>find_translation()</td>
<td>returns the translation vector between two translation of two polyhedron (only for Polyhedron over ZZ)</td>
</tr>
<tr>
<td>integrate()</td>
<td>computes the integral of a polynomial over the polyhedron</td>
</tr>
<tr>
<td>radius()</td>
<td>returns the radius of the smallest sphere containing the polyhedron</td>
</tr>
<tr>
<td>radius_square()</td>
<td>returns the square of the radius of the smallest sphere containing the polyhedron</td>
</tr>
<tr>
<td>volume()</td>
<td>computes different volumes of the polyhedron</td>
</tr>
<tr>
<td>restricted_automorphism_group()</td>
<td>returns the restricted automorphism group</td>
</tr>
<tr>
<td>lattice_automorphism_group()</td>
<td>returns the lattice automorphism group. Only for PPL Lattice Polytope</td>
</tr>
</tbody>
</table>

**Polyhedra tips and tricks**

*Author: Jean-Philippe Labbé <labbe@math.fu-berlin.de>*

**Operation shortcuts**

You can obtain different operations using natural symbols:

```sage
sage: Cube = polytopes.cube()
sage: Octahedron = 3/2*Cube.polar()  # Dilation
sage: Cube + Octahedron  # Minkowski sum
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 24 vertices
sage: Cube & Octahedron  # Intersection
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 24 vertices
sage: Cube * Octahedron  # Cartesian product
A 6-dimensional polyhedron in QQ^6 defined as the convex hull of 48 vertices
sage: Cube - Polyhedron(verts=[[-1,0,0],[1,0,0]])  # Minkowski difference
A 2-dimensional polyhedron in QQ^3 defined as the convex hull of 4 vertices
```

**Sage input function**

If you are working with a polyhedron that was difficult to construct and you would like to get back the proper Sage input code to reproduce this object, you can!

```sage
sage: Cube = polytopes.cube()
sage: TCube = Cube.truncation().dilation(1/2)
sage: sage_input(TCube)
Polyhedron(backend='ppl', base_ring=QQ, vertices=[[1/6, -1/2, -1/2],
(1/2, -1/6, -1/2), (1/2, 1/6, -1/2), (1/2, 1/2, -1/6), (1/2, 1/2, 1/6),
(1/2, 1/6, 1/2), (1/6, 1/2, 1/2), (1/2, -1/6, 1/2), (1/2, 1/2, -1/2),
(1/6, -1/2, 1/2), (1/2, -1/2, 1/6), (1/2, -1/2, -1/6), (-1/2, 1/6, -1/2),
(-1/2, -1/6, 1/2), (-1/2, -1/6, 1/2), (-1/2, -1/6, 1/2), (-1/6, -1/2, 1/2),
(-1/2, 1/2, 1/2), (-1/6, 1/2, -1/2), (-1/2, 1/2, -1/2), (-1/6, -1/2, 1/2),
(-1/2, -1/2, -1/6), (-1/6, -1/2, -1/2), (-1/2, -1/6, -1/2)]
```

12.1. Thematic tutorial document tree
If you would like to visualize the $H$-representation nicely and even get the latex presentation, there is a method for that!

```python
sage: Nice_repr = TCube.Hrepresentation_str()
sage: print(Nice_repr)
-6*x0 - 6*x1 - 6*x2 >= -7
-6*x0 - 6*x1 + 6*x2 >= -7
-6*x0 + 6*x1 - 6*x2 >= -7
-6*x0 + 6*x1 + 6*x2 >= -7
  -2*x0 >= -1
  -2*x1 >= -1
  -2*x2 >= -1
6*x0 + 6*x1 + 6*x2 >= -7
  2*x2 >= -1
  2*x1 >= -1
  2*x0 >= -1
6*x0 - 6*x1 - 6*x2 >= -7
6*x0 - 6*x1 + 6*x2 >= -7
6*x0 + 6*x1 - 6*x2 >= -7
```

```python
sage: print(TCube.Hrepresentation_str(latex=True))
\begin{array}{rcl}
-6 x_{0} - 6 x_{1} - 6 x_{2} & \geq & -7 \\
-6 x_{0} - 6 x_{1} + 6 x_{2} & \geq & -7 \\
-6 x_{0} + 6 x_{1} - 6 x_{2} & \geq & -7 \\
-6 x_{0} + 6 x_{1} + 6 x_{2} & \geq & -7 \\
  -2 x_{0} & \geq & -1 \\
  -2 x_{1} & \geq & -1 \\
  -2 x_{2} & \geq & -1 \\
6 x_{0} + 6 x_{1} + 6 x_{2} & \geq & -7 \\
  2 x_{2} & \geq & -1 \\
  2 x_{1} & \geq & -1 \\
  2 x_{0} & \geq & -1 \\
6 x_{0} - 6 x_{1} - 6 x_{2} & \geq & -7 \\
6 x_{0} - 6 x_{1} + 6 x_{2} & \geq & -7 \\
6 x_{0} + 6 x_{1} - 6 x_{2} & \geq & -7 \\
\end{array}
```

```python
sage: Latex_repr = LatexExpr(TCube.Hrepresentation_str(latex=True))
sage: view(Latex_repr)  # not tested
```

The `style` parameter allows to change the way to print the $H$-relations:

```python
sage: P = polytopes.permutahedron(3)
sage: print(P.Hrepresentation_str(style='<='))
-x0 - x1 - x2 == -6
  -x0 - x1 <= -3
  x0 + x1 <= 5
  -x1 <= -1
  x0 <= 3
  -x0 <= -1
  x1 <= 3
```

(continues on next page)
Visualization of polyhedron objects in Sage

Author: sarah-marie belcastro <smbelcas@toroidalsnark.net>, Jean-Philippe Labbé <labbe@math.fu-berlin.de>

There are different ways to visualize polyhedron object of dimension at most 4.

**render_solid**

This plots the polyhedron as a solid. You can also adjust the opacity parameter.

```sage
Cube = polytopes.cube()
sage: Cube.render_solid(opacity=0.7)
```

**render_wireframe**

This plots the graph (with unbounded edges) of the polyhedron

```sage
Cube.render_wireframe()
```

**plot**

The `plot` method draws the graph, the polygons and vertices of the polyhedron all together.

```sage
Cube.plot()
```

**show**

This is similar to `plot` but does not return an object that you can manipulate.
schlegel_projection

It is possible to visualize 4-dimensional polytopes using a schlegel diagram.

```
sage: HC = polytopes.hypercube(4)
sage: HC.schlegel_projection()
The projection of a polyhedron into 3 dimensions
sage: HC.schlegel_projection().plot()
Graphics3d Object
```

We can see it from a different perspective by choosing point at a different distance:

```
sage: HC.schlegel_projection(position=1/4).plot()
Graphics3d Object
```

It is possible to choose from which facet one sees the projection:

```
sage: tHC = HC.face_truncation(HC.faces(0)[0])
sage: tHC.facets()
(A 3-dimensional face of a Polyhedron in QQ^4 defined as the convex hull of 10 vertices,
 A 3-dimensional face of a Polyhedron in QQ^4 defined as the convex hull of 10 vertices,
 A 3-dimensional face of a Polyhedron in QQ^4 defined as the convex hull of 10 vertices,
 A 3-dimensional face of a Polyhedron in QQ^4 defined as the convex hull of 10 vertices,
 A 3-dimensional face of a Polyhedron in QQ^4 defined as the convex hull of 4 vertices,
 A 3-dimensional face of a Polyhedron in QQ^4 defined as the convex hull of 8 vertices,
 A 3-dimensional face of a Polyhedron in QQ^4 defined as the convex hull of 8 vertices,
 A 3-dimensional face of a Polyhedron in QQ^4 defined as the convex hull of 8 vertices,
 A 3-dimensional face of a Polyhedron in QQ^4 defined as the convex hull of 8 vertices)
sage: tHC.schlegel_projection(tHC.facets()[4]).plot()
Graphics3d Object
```

tikz

This method returns a tikz picture of the polytope (must be 2 or 3-dimensional). For more detail see the tutorial *Draw polytopes in LaTeX using TikZ*.

```
sage: c = polytopes.cube()
sage: c.tikz().splitlines()[:5]
['\\begin{tikzpicture}%',
 '\t[x=\{(1.000000cm, 0.000000cm)\},',
 '\t[y=\{(-0.000000cm, 1.000000cm)\},',
 '\t[z=\{(0.000000cm, -0.000000cm)\},',
 '\tscale=1.000000,']
```

Draw polytopes in LaTeX using TikZ

Author: Jean-Philippe Labbé <labbe@math.fu-berlin.de>

It is sometimes very helpful to draw 3-dimensional polytopes in a paper. TikZ is a very versatile tool to draw in scientific documents and Sage can deal easily with 3-dimensional polytopes. Finally sagetex makes everything work together nicely between Sage, TikZ and LaTeX. Since version 6.3 of Sage, there is a function for (projection of) polytopes to output a TikZ picture of the polytope. This short tutorial shows how it all works.

Instructions

To put an image of a 3D-polytope in LaTeX using TikZ and Sage, simply follow the instructions:

- Install SageTex (optional but recommended!)
- Put \usepackage{tikz} in the preamble of your article
- Open Sage and change the directory to your article’s by the command cd /path/to/article
- Input your polytope, called P for example, to Sage
- Visualize the polytope P using the command P.show(aspect_ratio=1)
- This will open an interactive view in your default browser, where you can rotate the polytope.
- Once the desired view angle is found, click on the information icon in the lower right-hand corner and select Get Viewpoint. This will copy a string of the form \[x,y,z\text{,angle}\] to your local clipboard.
- Go back to Sage and type Img = P.tikz([x,y,z],angle). You can paste the string here to save some typing.
- Img now contains a Sage object of type LatexExpr containing the raw TikZ picture of your polytope

Then, you can either copy-paste it to your article by typing Img in Sage or save it to a file, by doing

```python
f = open('Img_poly.tex','w')
f.write(Img)
f.close()
```

Then in the pwd (present working directory of sage, the one of your article) you will have a file named Img_poly.tex containing the tikzpicture of your polytope.

Customization

You can customize the polytope using the following options in the command P.tikz()

- scale: positive number to scale the polytope,
- edge_color: string (default: blue!95!black) representing colors which tikz recognize,
- facet_color: string (default: blue!95!black) representing colors which tikz recognize,
- vertex_color: string (default: green) representing colors which tikz recognize,
- opacity: real number (default: 0.8) between 0 and 1 giving the opacity of the front facets,
- axis: Boolean (default: False) draw the axes at the origin or not.
Examples

Let's say you want to draw the polar dual of the following (nice!) polytope given by the following list of vertices:

\[[1,0,1],[1,0,0],[1,1,0],[0,0,-1],[0,1,0],[-1,0,0],[0,1,1],[0,0,1],[0,-1,0]\]

In Sage, you type:

```python
P = Polyhedron(vertices=[[1,0,1],[1,0,0],[1,1,0],[0,0,-1],[0,1,0],[-1,0,0],[0,1,1],[0,0,1],[0,-1,0]]).polar()
```

Then, you visualize the polytope by typing `P.show(aspect_ratio=1)`

When you found a good angle, follow the above procedure to obtain the values \([674,108,-731]\) and angle=112, for example.

```python
Img = P.tikz([674,108,-731],112)
```

Or you may want to customize using the command

```python
Img = P.tikz([674,108,-731],112, scale=2, edge_color='orange', facet_color='red', vertex_color='blue', opacity=0.4)
```

Further, you may want to edit deeper the style of the polytope, directly inside the tikzpicture. For example, line 6-9 in the tikzpicture reads:

```latex
back/.style={loosely dotted, thin},
edge/.style={color=orange, thick},
facet/.style={fill=red, fill opacity=0.400000},
vertex/.style={inner sep=1pt, circle, draw=blue!25!black, fill=blue!75!black, thick, anchor=base}
```

It is also possible to replace it by the following 4 lines (and adding `\usetikzlibrary{shapes}` in the preamble)

```latex
back/.style={loosely dashed, line width=2pt},
edge/.style={color=yellow, line width=2pt},
facet/.style={fill=cyan, fill opacity=0.400000},
vertex/.style={inner sep=1pt, star, star points=7, draw=blue!75!white, fill=blue!85!white, thick, anchor=base}
```

Finally, you may want to tweak your picture by adding labels, elements on vertices, edges, facets, etc.

**Automatize using SageTex**

For this you need to put

\texttt{\usepackage{sagetex}}

in the preamble of your article

There are different ways to use sagetex and you may create your own. Here are some possibilities.

1) You can directly type in a sagestr in the article:

```latex
\sagestr{(polytopes.permutahedron(4)).tikz([4,5,6],45, scale=0.75, facet\_color='red', \_vertex\_color='yellow', opacity=0.3)}
```
2) You may create the following tex commands

\begin{verbatim}
newcommand\polytopeimg[4]{\sagestr{(#1).tikz(#2,#3,#4)}}
newcommand\polytopeimgopt[9]{\sagestr{(#1).tikz(#2,#3,#4,#5,#6,#7,#8,#9)}}
\end{verbatim}

in your preamble and use them with a \texttt{sagesilent} in your article:

\begin{verbatim}
\begin{sagesilent}
Polytope = polytopes.great_rhombicuboctahedron()
\end{sagesilent}
\polytopeimg{Polytope}{[276,-607,-746]}{102}{1}
\polytopeimgopt{Polytope}{view=[-907,379,183]}{angle=129}{scale=2}{edge_color='red'}
{facet_color='yellow'}{vertex_color='blue'}{opacity=0.6}{axis=False}
\end{verbatim}

Then, run pdflatex, execute Sage on the file \texttt{article_name.sagetex.sage} and run pdflatex again.

For more information on SageTeX, see the tutorial \url{http://doc.sagemath.org/html/en/tutorial/sagetex.html}.

\section*{12.1.11 Steenrod Algebra Modules}

\textit{Author: Robert R. Bruner, Michael J. Catanzaro, Sverre Lunoe-Nielsen, and Koen van Woerden}

Let $p$ be a prime number. The mod $p$ Steenrod algebra $A$ is a connected algebra over $\mathbb{F}_p$, the finite field of $p$ elements. All modules presented here will be defined over $A$ or one of its sub-Hopf algebras. E.g.:

\begin{verbatim}
sage: A = SteenrodAlgebra(p=2)
\end{verbatim}

The constructor of the module class takes as arguments an ordered tuple of degrees and the algebra over which the module is defined, together with an optional set of relations:

\begin{verbatim}
sage: from sage.modules.fp_graded.steenrod.module import SteenrodFPModule
sage: F = SteenrodFPModule(A, [0, 1, 7]); F
\end{verbatim}

Finitely presented left module on 3 generators over mod 2 Steenrod algebra, milnor basis

Denote the module generators of an $A$-module $M$ by $g_{d_1}, \ldots, g_{d_N}$, where subscripts denote their degrees. A homogeneous relation of degree $n$ has the form

$$\sum_{i=1}^{N} a_i \cdot g_{d_i} = 0,$$

where the homogeneous coefficients $a_1, \ldots, a_N$ lie in $A$, such that $\deg(a_i) + \deg(g_{d_i}) = n$ for $i = 1, \ldots, N$. To create a module with relations, the coefficients for each relation is given:

\begin{verbatim}
sage: r1 = [Sq(8), Sq(7), 0]  # First relation
sage: r2 = [Sq(7), 0, 1]     # Second relation
sage: M = SteenrodFPModule(A, [0, 1, 7], relations=[r1, r2]); M
\end{verbatim}

Finitely presented left module on 3 generators and 2 relations over mod 2 Steenrod algebra, milnor basis

The resulting module will have three generators in the degrees we gave them:

\begin{verbatim}
sage: M.generator_degrees()
(0, 1, 7)
\end{verbatim}
The connectivity of a module over a connected graded algebra is the minimum degree of all its module generators. Thus, if the module is non-trivial, the connectivity is an integer:

```
sage: M.connectivity()
0
```

Each module is defined over a Steenrod algebra or some sub-Hopf algebra of it, given by its base ring:

```
sage: M.base_ring()
mod 2 Steenrod algebra, milnor basis
sage: SteenrodFPModule(SteenrodAlgebra(p=2,profile=(3,2,1)), [0]).base_ring()
sub-Hopf algebra of mod 2 Steenrod algebra, milnor basis, profile function
[3, 2, 1]
```

**Note:** Calling `algebra()` will not return the desired algebra. Users should use the `base_ring()` method.

### Module elements

Module elements are displayed in terms of generators, which by default are called `g[degree]`:

```
sage: M.an_element(n=5)
Sq(2,1)*g[0] + Sq(4)*g[1]
sage: e = M.an_element(n=15); e
Sq(0,0,0,1)*g[0] + Sq(1,2,1)*g[1] + Sq(8)*g[7]
sage: e.dense_coefficient_list()
[Sq(0,0,0,1), Sq(1,2,1), Sq(8)]
```

The generators are themselves elements of the module:

```
sage: gens = M.generators(); gens
(g[0], g[1], g[7])
sage: gens[0] in M
True
```

One can produce an element from a given set of algebra coefficients:

```
sage: coeffs = [Sq(15), Sq(10)*Sq(1,1), Sq(8)]
sage: x = M(coeffs); x
Sq(15)*g[0] + (Sq(4,1,1)+Sq(7,0,1)+Sq(11,1))*g[1] + Sq(8)*g[7]
```

The module action produces new elements:

```
sage: Sq(2) * x
Sq(14,1)*g[0] + (Sq(7,1)+Sq(10))g[7]
```

Each non-zero homogeneous element has a well-defined degree:

```
sage: x.degree()
15
```

but the zero element does not:
sage: zero = M.zero(); zero
0
sage: zero.degree()
Traceback (most recent call last):
...
ValueError: the zero element does not have a well-defined degree

At this point it may be useful to point out that elements are not reduced to a minimal representation when they are created. A normalization can be enforced, however:

sage: g7 = M([0, 0, 1]); g7
g[7]
sage: g7.normalize()
Sq(7)*g[0]
sage: g7 == g7.normalize()
True
sage: m = M([Sq(7), 0, 0])
sage: s = m + g7; s  # m and g7 are related by m = Sq(7)*g[0] = g[7],
Sq(7)*g[0] + g[7]
sage: s == 0  # so their sum should zero.
True
sage: s.normalize()  # Its normalized form is more revealing.
0

For every integer \( n \), the set of module elements of degree \( n \) form a vector space over the ground field \( \mathbb{F}_p \). A basis for this vector space can be computed:

sage: M.basis_elements(7)
(Sq(0,0,1)*g[0],
 Sq(1,2)*g[0],
 Sq(4,1)*g[0],
 Sq(7)*g[0],
 Sq(0,2)*g[1],
 Sq(3,1)*g[1],
 Sq(6)*g[1])

Note that the third generator \( g_7 \) of degree 7 is apparently missing from the basis above. This is because of the relation \( Sq^7(g_0) = g_7 \).

A vector space presentation can be produced:

sage: M.vector_presentation(5)
Vector space quotient V/W of dimension 4 over Finite Field of size 2 where
V: Vector space of dimension 4 over Finite Field of size 2
W: Vector space of degree 4 and dimension 0 over Finite Field of size 2
Basis matrix:
[]

Given any element, its coordinates with respect to this basis can be computed:

sage: x = M.an_element(7); x
Sq(0,0,1)*g[0] + Sq(3,1)*g[1] + g[7]
Going the other way, any element can be constructed by specifying its coordinates:

```
sage: x_ = M.element_from_coordinates((1, 0, 0, 1, 0, 1, 0), 7)
sage: x_
(Sq(0,0,1)+Sq(7))*g[0] + Sq(3,1)*g[1]
sage: x_ == x
True
```

### Module homomorphisms

Homomorphisms of $A$-modules $M \rightarrow N$ are linear maps of their underlying $\mathbb{F}_p$-vector spaces which commute with the $A$-module structure. Homomorphisms are required to be homogeneous but need not be degree zero.

To create a homomorphism, first create the object modeling the set of all such homomorphisms using the function `Hom`:

```
sage: Hko = SteenrodFPModule(A, [0], [[Sq(2)], [Sq(1)]]
sage: homspace = Hom(Hko, Hko); homspace
Set of Morphisms from Finitely presented left module on 1 generator and 2 relations
over mod 2 Steenrod algebra, milnor basis
to Finitely presented left module on 1 generator and 2 relations
over mod 2 Steenrod algebra, milnor basis
in Category of finitely presented graded modules over mod 2 Steenrod algebra, milnor basis
```

Just as with module elements, homomorphisms are created using the homspace. The only argument is a list of elements in the codomain, giving the images of the module generators of the domain:

```
sage: gen = Hko.generator(0)  # the generator of the codomain module
sage: values = [Sq(0, 0, 1) * gen]; values
[Sq(0,0,1)*g[0]]
sage: f = homspace(values)
```

The resulting homomorphism is the one sending the $i$-th generator of the domain to the $i$-th codomain value given:

```
sage: f
Module endomorphism of Finitely presented left module on 1 generator and 2 relations
over mod 2 Steenrod algebra, milnor basis
Defn: g[0] |--> Sq(0,0,1)*g[0]
```

Homomorphisms can be evaluated on elements of the domain module:

```
sage: v1 = f(Sq(4)*gen); v1
Sq(4,0,1)*g[0]
sage: v2 = f(Sq(2)*Sq(4)*gen); v2
(Sq(3,1,1)+Sq(6,0,1))*g[0]
```

and they respect the module action:
Convenience methods exist for creating the trivial morphism:

```python
sage: x = Sq(4)*Sq(7)*gen
sage: x == 0
False
sage: zero_map = homspace.zero(); zero_map
Module endomorphism of Finitely presented left module on 1 generator and 2 relations...
...over mod 2 Steenrod algebra, milnor basis
  Defn: g[0] |--> 0
sage: zero_map(x)
0
sage: zero_map(x).is_zero()
True
```

as well as the identity endomorphism:

```python
sage: one = Hom(Hko, Hko).identity(); one
Module endomorphism of Finitely presented left module on 1 generator and 2 relations...
...over mod 2 Steenrod algebra, milnor basis
  Defn: g[0] |--> g[0]
sage: one.is_endomorphism()
True
sage: one.is_identity()
True
```

Any non-trivial homomorphism has a well defined degree:

```python
sage: f.degree()
7
```

but just as for module elements, the trivial homomorphism does not:

```python
sage: zero_map = homspace.zero()
sage: zero_map.degree()
Traceback (most recent call last):
  ...
ValueError: the zero morphism does not have a well-defined degree
```

Any two homomorphisms can be added as long as they are of the same degree:

```python
sage: f1 = homspace(Hko([Sq(0,0,3) + Sq(0,2,0,1)]))
sage: f2 = homspace(Hko([Sq(8,2,1)]))
sage: (f1 + f2).is_zero()
False
sage: f1 + f2
(continues on next page)
```
Module endomorphism of Finitely presented left module on 1 generator and 2 relations
→ over mod 2 Steenrod algebra, milnor basis
Defn: g[0] |--> (Sq(0,0,3)+Sq(0,2,0,1)+Sq(8,2,1))*g[0]

or when at least one of them is zero:

```python
sage: f + zero_map == f
True
```

but not if they have different degrees:

```python
sage: F = SteenrodFPModule(A, [0])
sage: b4 = Hom(F, F)([Sq(4) * F.generator(0)])
sage: b8 = Hom(F, F)([Sq(8) * F.generator(0)])
sage: b4 + b8
Traceback (most recent call last):
...
ValueError: morphisms do not have the same degree
```

Finally, additive inverses exist:

```python
sage: (f - f) == 0
True
```

The restriction of a homomorphism to the vector space of \(n\)-dimensional module elements is a linear transformation:

```python
sage: f_21 = f.vector_presentation(21); f_21
Vector space morphism represented by the matrix:
[1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0]
[0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0]
[0 0 0 1 0 0 0 0 0 0 1 0 0 1 0 0 1 0 0 1]
[0 0 0 0 1 0 0 0 0 0 0 1 0 0 1 0 0 1 0 0 1]
[0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 1 0 0 1 0 0 1]
[0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 1 0 0 1 0 0 1]
[0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 1 0 0 1 0 0 1]
[0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 1 0 0 1 0 0 1]
[0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 1 0 0 1 0 0 1]
[0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 1 0 0 1 0 0 1]
[0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 1 0 0 1 0 0 1]
[0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 1 0 0 1 0 0 1]
[0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 1 0 0 1 0 0 1]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 1 0 0 1 0 0 1]
Codomain: Vector space quotient V/W of dimension 6 over Finite Field of size 2 where
```

(continues on next page)
V: Vector space of dimension 35 over Finite Field of size 2
W: Vector space of degree 35 and dimension 29 over Finite Field of size 2
Basis matrix:
29 x 35 dense matrix over Finite Field of size 2

This is compatible with the vector presentations of its domain and codomain modules:

```sage
sage: f.domain() is Hko
True
sage: f.codomain() is Hko
True
sage: f_21.domain() is Hko.vector_presentation(21)
True
sage: f_21.codomain() is Hko.vector_presentation(21 + f.degree())
True
```

Elements in the preimage of a homomorphism can be found:

```sage
sage: f.solve(Sq(2)*Sq(4)*Sq(7)*gen)
Sq(0,2)*g[0]
sage: f.solve(Sq(8)*gen) is None
True
```

Homomorphisms can be composed as expected:

```sage
sage: g = homspace([Sq(0, 0, 0, 1)*gen]); g
Module endomorphism of Finitely presented left module on 1 generator and 2 relations over mod 2 Steenrod algebra, milnor basis
  Defn: g[0] |--> Sq(0,0,0,1)*g[0]
sage: g*f
Module endomorphism of Finitely presented left module on 1 generator and 2 relations over mod 2 Steenrod algebra, milnor basis
  Defn: g[0] |--> Sq(0,0,1,1)*g[0]
sage: one = homspace.identity()
sage: f*one == f
True
```

**Homological algebra**

The category of modules over $A$ is Abelian, so kernels, images and cokernels all exist and can be computed using the morphisms.

**Note:** Methods for morphisms like
- `kernel_inclusion()`,
- `cokernel_projection()`,
- `image()`.  

12.1. Thematic tutorial document tree
• *homology()*

compute sub- and quotient modules related to homomorphisms, but they do not return instances of the module class. Rather, they return the natural homomorphisms which connect these modules to the modules that gave rise to them.

For example the function *kernel_inclusion()* returns an injective homomorphism which is onto the kernel submodule we asked it to compute, while the function *cokernel_projection()* provides a surjective homomorphism onto the cokernel module.

In each case, getting a reference to the module instance requires calling *domain()* or *codomain()* on the returned homomorphism, depending on the case.

Refer to each function’s documentation for specific details.

### Cokernels

In the following example, we define a cyclic module $HZ$ with one relation in two ways: first explicitly, and then as the cokernel of a homomorphism of free modules. We then construct a candidate for an isomorphism and check that it is both injective and surjective:

```python
sage: HZ = SteenrodFPModule(A, [0], [[Sq(1)]]); HZ
Finitely presented left module on 1 generator and 1 relation over mod 2 Steenrod algebra, → milnor basis

sage: F = SteenrodFPModule(A, [0])
sage: j = Hom(F, F)([Sq(1)*F.generator(0)])
sage: coker = j.cokernel_projection() # the natural quotient homomorphism onto the_cokernel.
sage: hz = coker.codomain(); hz
Finitely presented left module on 1 generator and 1 relation over mod 2 Steenrod algebra, → milnor basis

sage: a = Hom(HZ, hz)([hz.generator(0)])
sage: a.is_injective()
True
sage: a.is_surjective()
True
```

### Kernels

When computing the kernel of a homomorphism $f$, the result is an injective homomorphism into the domain of $f$:

```python
sage: k = f.kernel_inclusion(); k
Module morphism:
  From: Finitely presented left module on 1 generator and 3 relations over mod 2, → Steenrod algebra, milnor basis
  To: Finitely presented left module on 1 generator and 2 relations over mod 2, → Steenrod algebra, milnor basis
  Defn: g[7] |--> Sq(0,0,1)*g[0]
sage: k.codomain() == f.domain()
True
sage: k.is_injective()
```

(continues on next page)
We can check that the injective image of $k$ is the kernel of $f$ by showing that $f$ factors as $h \circ c$, where $c$ is the quotient map to the cokernel of $k$, and $h$ is injective:

```python
sage: K = k.codomain()  # We want to check that this really is the kernel of f.
sage: coker = k.cokernel_projection()  # coker is the natural map: Hko -> coker(f).
˓→ with kernel K.
sage: h = Hom(coker.codomain(), Hko)(f.values())
sage: h*coker == f  # Is K contained in ker(f) ?
True
sage: h.is_injective()  # Is ker(f) contained in K ?
True
```

### Images

The method `image()` behaves similarly, returning an injective homomorphism with image equal to the submodule $\text{im}(f)$:

```python
sage: i = f.image(); i
Module morphism:
   From: Finitely presented left module on 1 generator and 3 relations over mod 2 Steenrod algebra, milnor basis
   To:   Finitely presented left module on 1 generator and 2 relations over mod 2 Steenrod algebra, milnor basis
Defn: g[7] |--> Sq(0,0,1)*g[0]
sage: i.codomain() == f.codomain()
True
sage: i.is_injective()
True
```

We can check that the injective image of $i$ is the image of $f$ by lifting $f$ over $i$, and showing that the lift is surjective:

```python
sage: f_ = f.lift(i); f_
Module morphism:
   From: Finitely presented left module on 1 generator and 2 relations over mod 2 Steenrod algebra, milnor basis
   To:   Finitely presented left module on 1 generator and 3 relations over mod 2 Steenrod algebra, milnor basis
Defn: g[0] |--> g[7]
sage: i*f_ == f  # Is im(i) contained in im(f) ?
True
sage: f_.is_surjective()  # Is im(f) contained in im(i) ?
True
```

When a pair of composable homomorphisms $g \circ f : M \to N \to L$ satisfy the condition $g \circ f = 0$, the sub-quotient $\text{ker}(g)/\text{im}(f)$ can be computed and is given by the natural quotient homomorphism with domain $\text{ker}(g)$:

12.1. Thematic tutorial document tree

---

```
True
sage: ker = k.domain()
sage: ker
Finitely presented left module on 1 generator and 3 relations over mod 2 Steenrod algebra, milnor basis
We can check that the injective image of k is the kernel of f by showing that f factors as h o c, where c is the quotient map to the cokernel of k, and h is injective:

```

```

```python
sage: K = k.codomain()  # We want to check that this really is the kernel of f.
sage: coker = k.cokernel_projection()  # coker is the natural map: Hko -> coker(f).
˓→ with kernel K.
sage: h = Hom(coker.codomain(), Hko)(f.values())
sage: h*coker == f  # Is K contained in ker(f) ?
True
sage: h.is_injective()  # Is ker(f) contained in K ?
True
```

### Images

The method `image()` behaves similarly, returning an injective homomorphism with image equal to the submodule $\text{im}(f)$:

```
```

```python
sage: i = f.image(); i
Module morphism:
   From: Finitely presented left module on 1 generator and 3 relations over mod 2 Steenrod algebra, milnor basis
   To:   Finitely presented left module on 1 generator and 2 relations over mod 2 Steenrod algebra, milnor basis
Defn: g[7] |--> Sq(0,0,1)*g[0]
sage: i.codomain() == f.codomain()
True
sage: i.is_injective()
True
```

We can check that the injective image of i is the image of f by lifting f over i, and showing that the lift is surjective:

```
```

```python
sage: f_ = f.lift(i); f_
Module morphism:
   From: Finitely presented left module on 1 generator and 2 relations over mod 2 Steenrod algebra, milnor basis
   To:   Finitely presented left module on 1 generator and 3 relations over mod 2 Steenrod algebra, milnor basis
Defn: g[0] |--> g[7]
sage: i*f_ == f  # Is im(i) contained in im(f) ?
True
sage: f_.is_surjective()  # Is im(f) contained in im(i) ?
True
```

When a pair of composable homomorphisms $g \circ f : M \to N \to L$ satisfy the condition $g \circ f = 0$, the sub-quotient $\text{ker}(g)/\text{im}(f)$ can be computed and is given by the natural quotient homomorphism with domain $\text{ker}(g)$:

12.1. Thematic tutorial document tree

---
**Thematic Tutorials, Release 9.7**

```python
sage: f * f == 0  # Does the kernel of f contain the image of f ?
True
sage: K = f.kernel_inclusion()  # k: ker(f) -> Hko
sage: h = f.homology(f)  # h: ker(f) -> ker(f) / im(f)
sage: h.codomain()  # This is the homology module.
```

Finitely presented left module on 1 generator and 4 relations over mod 2 Steenrod algebra, milnor basis

### Free resolutions

Finally, free resolutions can be computed. These calculations usually take some time to complete, so it is usually a good idea to raise the verbose flag to output progress information.

The following examples are taken from Michael Catanzaro’s thesis where the first version of this software appeared:

```python
sage: res = Hko.resolution(6, verbose=True)
Computing f_1 (1/6)
Computing f_2 (2/6)
Computing using the profile:
(2, 1)
Resolving the kernel in the range of dimensions [1, 8]: 1 2 3 4 5 6 7 8.
Computing f_3 (3/6)
Computing using the profile:
(2, 1)
Resolving the kernel in the range of dimensions [2, 10]: 2 3 4 5 6 7 8 9 10.
Computing f_4 (4/6)
Computing using the profile:
(2, 1)
Resolving the kernel in the range of dimensions [3, 13]: 3 4 5 6 7 8 9 10 11 12 13.
Computing f_5 (5/6)
Computing using the profile:
(2, 1)
Resolving the kernel in the range of dimensions [4, 18]: 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18.
Computing f_6 (6/6)
Computing using the profile:
(2, 1)
Resolving the kernel in the range of dimensions [5, 20]: 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20.
```

The result of the calculation is a list of all the maps in the resolution:

```python
sage: [f.domain() for f in res]
[Free graded left module on 1 generator over mod 2 Steenrod algebra, milnor basis,
Free graded left module on 2 generators over mod 2 Steenrod algebra, milnor basis,
Free graded left module on 2 generators over mod 2 Steenrod algebra, milnor basis,
Free graded left module on 2 generators over mod 2 Steenrod algebra, milnor basis,
Free graded left module on 3 generators over mod 2 Steenrod algebra, milnor basis,
Free graded left module on 3 generators over mod 2 Steenrod algebra, milnor basis,
Free graded left module on 4 generators over mod 2 Steenrod algebra, milnor basis,
Free graded left module on 4 generators over mod 2 Steenrod algebra, milnor basis]
```

```python
sage: def is_complex(res):
    (continues on next page)
```
for i in range(len(res)-1):
    f = (res[i]*res[i+1])
    if not f.is_zero():
        return False
    return True

sage: is_complex(res)
True

sage: def is_exact(res):
    for i in range(len(res)-1):
        h = res[i].homology(res[i+1])
        if not h.codomain().is_trivial():
            return False
        return True

sage: is_exact(res)
True

sage: [r.codomain().generator_degrees() for r in res]
[(0,), (0,), (2, 1), (2, 4), (3, 7), (4, 8, 12), (5, 9, 13, 14)]

sage: [r.values() for r in res]
[(g[0],),
 (Sq(2)*g[0], Sq(1)*g[0]),
 (Sq(1)*g[1], Sq(3)*g[1] + Sq(2)*g[2]),
 (Sq(1)*g[2], Sq(2,1)*g[2] + Sq(3)*g[4]),
 (Sq(1)*g[3], Sq(2,1)*g[3] + Sq(1)*g[7], Sq(2,1)*g[7]),
 (Sq(1)*g[4],
 Sq(2,1)*g[4] + Sq(1)*g[8],
 Sq(2,1)*g[8] + Sq(1)*g[12],
 Sq(2)*g[12]),
 (Sq(1)*g[5],
 Sq(2,1)*g[5] + Sq(1)*g[9],
 Sq(2,1)*g[9] + Sq(1)*g[13],
 Sq(0,1)*g[13] + Sq(2)*g[14])]}

Example: Counting lifts

In this more elaborate example we show how to find all possible lifts of a particular homomorphism. We will do this in two ways, and as a check of validity, we will compare the results in the end.

We will work with the following modules over the mod 2 Steenrod algebra $A$:

\[ H\mathbb{Z} = A/A \cdot Sq(1) \]  \hspace{1cm} (12.1)

\[ Hko = A/A \cdot \{Sq(2), Sq(1)\} \]  \hspace{1cm} (12.2)

There is a natural projection $q : H\mathbb{Z} \to Hko$, and a non-trivial endomorphism of degree 28, represented as a degree zero map $f : \Sigma^{28}Hko \to Hko$ that we define below.

The problem we will solve is to find all possible homomorphisms $f' : \Sigma^{28}Hko \to H\mathbb{Z}$, making the following diagram into a commuting triangle:

\[ H\mathbb{Z} \overset{q}{\longrightarrow} Hko \overset{f}{\longleftarrow} \Sigma^{28}Hko. \]
We begin by defining the modules and the homomorphisms \( f \) and \( q \). In the following, we let \( L = \Sigma^{28} Hko \):

```
sage: from sage.modules.fp_graded.steenrod.module import SteenrodFPModule
sage: A = SteenrodAlgebra(2)
sage: Hko = SteenrodFPModule(A, [0], [[Sq(2)],[Sq(1)]]),
sage: HZ = SteenrodFPModule(A, [0], [[Sq(1)]]),
sage: L = Hko.suspension(28)
```

The projection:

```
sage: q = Hom(HZ, Hko)([Hko.generator(0)])
sage: q
Module morphism:
   From: Finitely presented left module on 1 generator and 1 relation over mod 2 Steenrod algebra, milnor basis
   To:   Finitely presented left module on 1 generator and 2 relations over mod 2 Steenrod algebra, milnor basis
   Defn: g[0] |--> g[0]
```

The map to lift over \( q \):

```
sage: f = Hom(L, Hko)([Sq(0,2,1,1)*Hko.generator(0)])
sage: f
Module morphism:
   From: Finitely presented left module on 1 generator and 2 relations over mod 2 Steenrod algebra, milnor basis
   To:   Finitely presented left module on 1 generator and 2 relations over mod 2 Steenrod algebra, milnor basis
   Defn: g[28] |--> Sq(0,2,1,1)*g[0]
```

We will count the number of different lifts in two ways. First, we will simply compute the vector space of all possible maps \( L \to HZ \), and then check which of those become \( f \) when composed with \( q \):

```
sage: basis = Hom(L, HZ).basis_elements(0)  # The basis for the vector space of degree 0 maps L -> HZ
sage: from itertools import product
sage: def from_coords(c):
....:     '''Create a linear combination of the three basis homomorphisms.'''
....:     return c[0]*basis[0] + c[1]*basis[1] + c[2]*basis[2]

sage: for coords in product([0,1], repeat=3):
....:     print("%s: %s" % (coords, q*from_coords(coords) == f))
(0, 0, 0): False
(0, 0, 1): False
(0, 1, 0): True
(0, 1, 1): True
(1, 0, 0): True
(1, 0, 1): True
```

(continues on next page)
From this we conclude that four out of eight different homomorphisms $L \rightarrow H\mathbb{Z}$ are lifts of $f$:

\begin{verbatim}
sage: lifts = [from_coords((0,1,0)),
            ..., from_coords((0,1,1)),
            ..., from_coords((1,0,0)),
            ..., from_coords((1,0,1))]

sage: lifts

[Module morphism:
  From: Finitely presented left module on 1 generator and 2 relations over mod 2
  → Steenrod algebra, milnor basis
  To:   Finitely presented left module on 1 generator and 1 relation over mod 2
  → Steenrod algebra, milnor basis
  Defn: g[28] |--> Sq(6,5,1)*g[0],

  Module morphism:
  From: Finitely presented left module on 1 generator and 2 relations over mod 2
  → Steenrod algebra, milnor basis
  To:   Finitely presented left module on 1 generator and 1 relation over mod 2
  → Steenrod algebra, milnor basis
  Defn: g[28] |--> (Sq(6,5,1)+Sq(18,1,1))*g[0],

  Module morphism:
  From: Finitely presented left module on 1 generator and 2 relations over mod 2
  → Steenrod algebra, milnor basis
  To:   Finitely presented left module on 1 generator and 1 relation over mod 2
  → Steenrod algebra, milnor basis
  Defn: g[28] |--> Sq(10,1,0,1)*g[0],

  Module morphism:
  From: Finitely presented left module on 1 generator and 2 relations over mod 2
  → Steenrod algebra, milnor basis
  To:   Finitely presented left module on 1 generator and 1 relation over mod 2
  → Steenrod algebra, milnor basis
  Defn: g[28] |--> (Sq(10,1,0,1)+Sq(18,1,1))*g[0]]
\end{verbatim}

Alternatively we can use left-exactness of the functor $\text{Hom}_A(L, -)$ to enumerate all possible lifts of $f$. Start by finding a single lift of $f$ over the projection $q$:

\begin{verbatim}

sage: fl = f.lift(q); fl

Module morphism:
  From: Finitely presented left module on 1 generator and 2 relations over mod 2
  → Steenrod algebra, milnor basis
  To:   Finitely presented left module on 1 generator and 1 relation over mod 2
  → Steenrod algebra, milnor basis
  Defn: g[28] |--> (Sq(4,3,0,1)+Sq(6,0,1,1)+Sq(7,2,0,1)+Sq(10,1,0,1))*g[0]

We verify that $fl$ is indeed a lift:

\begin{verbatim}

sage: q * fl == f

True
\end{verbatim}

There is an exact sequence

\[ 0 \rightarrow \text{Hom}_A(L, \ker(q)) \xrightarrow{i_K} \text{Hom}_A(L, H\mathbb{Z}) \xrightarrow{q} \text{Hom}_A(L, Hko), \]
which means that the indeterminacy of choosing a lift for $f \in \text{Hom}_A(L, Hko)$ is represented by an element in $\text{Hom}_A(L, \ker(q))$. Therefore, we can proceed to count the number of lifts by computing this vector space of homomorphisms:

```python
sage: iK = q.kernel_inclusion()
sage: K = iK.domain()
sage: K.generator_degrees()
(2,)
sage: K.relations()
(Sq(2)*g[2],)
sage: ind = Hom(L, K).basis_elements(0); len(ind)
2
```

So now we know that the vector space of indeterminacies is 2-dimensional over the field of two elements. This means that there are four distinct lifts of $f$ over $q$, and we can construct these by taking the one lift we already found, and add to it all the different elements in the image of $iK_z$:

```python
sage: flift = [fl,
    ....: fl + iK * ind[0],
    ....: fl + iK * ind[1],
    ....: fl + iK * (ind[0] + ind[1])
]
sage: flift
[Module morphism:
  From: Finitely presented left module on 1 generator and 2 relations over mod 2
  ←→ Steenrod algebra, milnor basis
  To:   Finitely presented left module on 1 generator and 1 relation over mod 2
  ←→ Steenrod algebra, milnor basis
  Defn: g[28] |--> (Sq(4,3,0,1)+Sq(6,0,1,1)+Sq(7,2,0,1)+Sq(10,1,0,1))*g[0],
Module morphism:
  From: Finitely presented left module on 1 generator and 2 relations over mod 2
  ←→ Steenrod algebra, milnor basis
  To:   Finitely presented left module on 1 generator and 1 relation over mod 2
  ←→ Steenrod algebra, milnor basis
  Defn: g[28] |--> (Sq(0,7,1)+Sq(3,6,1)+Sq(4,1,3)+Sq(6,0,1,1)+Sq(6,5,1)+Sq(7,0,3))*g[0],
Module morphism:
  From: Finitely presented left module on 1 generator and 2 relations over mod 2
  ←→ Steenrod algebra, milnor basis
  To:   Finitely presented left module on 1 generator and 1 relation over mod 2
  ←→ Steenrod algebra, milnor basis
  Defn: g[28] |--> (Sq(4,3,0,1)+Sq(6,0,1,1)+Sq(7,2,0,1)+Sq(10,1,0,1)+Sq(12,3,1)+Sq(15,2,3,1)+Sq(18,1,1))*g[0],
Module morphism:
  From: Finitely presented left module on 1 generator and 2 relations over mod 2
  ←→ Steenrod algebra, milnor basis
  To:   Finitely presented left module on 1 generator and 1 relation over mod 2
  ←→ Steenrod algebra, milnor basis
  Defn: g[28] |--> (Sq(0,7,1)+Sq(3,6,1)+Sq(4,1,3)+Sq(6,0,1,1)+Sq(6,5,1)+Sq(7,0,3)+Sq(12,3,1)+Sq(15,2,3,1)+Sq(18,1,1))*g[0]]
```

As a test of correctness, we now compare the two sets of lifts. As they stand, it is not obvious that the lists `flift` and `lifts` are the same (up to a re-ordering of list elements), so the following comparison is reassuring:

```python
sage: flift[0] == lifts[2]
True
```
12.1.12 Tutorial: Programming in Python and Sage

Author: Florent Hivert <florent.hivert@univ-rouen.fr>, Franco Saliola <saliola@gmail.com>, et al.

This tutorial is an introduction to basic programming in Python and Sage, for readers with elementary notions of programming but not familiar with the Python language. It is far from exhaustive. For a more complete tutorial, have a look at the Python Tutorial. Also Python's documentation and in particular the standard library can be useful.

A more advanced tutorial presents the notions of objects and classes in Python.

Here are further resources to learn Python:

- Learn Python in 10 minutes ou en français Python en 10 minutes
- Dive into Python is a Python book for experienced programmers. Also available in other languages.
- Discover Python is a series of articles published in IBM’s developerWorks technical resource center.

**Data structures**

In Python, *typing is dynamic*; there is no such thing as declaring variables. The function `type()` returns the type of an object `obj`. To convert an object to a type `typ` just write `typ(obj)` as in `int("123")`. The command `isinstance(ex, typ)` returns whether the expression `ex` is of type `typ`. Specifically, any value is an instance of a class and there is no difference between classes and types.

The symbol `=` denotes the affectation to a variable; it should not be confused with `==` which denotes mathematical equality. Inequality is `!=`.

The standard types are `bool`, `int`, `list`, `tuple`, `set`, `dict`, `str`.

- The type `bool (booleans)` has two values: `True` and `False`. The boolean operators are denoted by their names `or`, and `not`.
- The Python types `int` and `long` are used to represent integers of limited size. To handle arbitrary large integers with exact arithmetic, Sage uses its own type named `Integer`.
- A `list` is a data structure which groups values. It is constructed using brackets as in `[1, 3, 4]`. The `range()` function creates integer lists. One can also create lists using list comprehension:

  ```python
  [ <expr> for <name> in <iterable> (if <condition>) ]
  ```

  For example:

  ```python
  sage: [ i^2 for i in range(10) if i % 2 == 0 ]
  [0, 4, 16, 36, 64]
  ```

- A `tuple` is very similar to a list; it is constructed using parentheses. The empty tuple is obtained by `()` or by the constructor `tuple`. If there is only one element, one has to write `(a,)`. A tuple is immutable (one cannot change it) but it is hashable (see below). One can also create tuples using comprehensions:
Thematic Tutorials, Release 9.7

```
sage: tuple(i^2 for i in range(10) if i % 2 == 0)
(0, 4, 16, 36, 64)
```

- A set is a data structure which contains values without multiplicities or order. One creates it from a list (or any iterable) with the constructor `set`. The elements of a set must be hashable:

```
sage: set([2,2,1,4,5])
{1, 2, 4, 5}
sage: set([[1], [2]])
Traceback (most recent call last):
  ...
TypeError: unhashable type: 'list'
```

- A dictionary is an association table, which associates values to keys. Keys must be hashable. One creates dictionaries using the constructor `dict`, or using the syntax:

```
{key1 : value1, key2 : value2 ...
```

For example:

```
sage: age = {'toto' : 8, 'mom' : 27}; age
{'mom': 27, 'toto': 8}
```

- Quotes (simple `' ' or double " ") enclose character strings. One can concatenate them using `+`.

- For lists, tuples, strings, and dictionaries, the indexing operator is written `l[i]`. For lists, tuples, and strings one can also uses slices as `l[:], l[:b], l[a:],` or `l[a:b]`. Negative indices start from the end.

- The `len()` function returns the number of elements of a list, a tuple, a set, a string, or a dictionary. One writes `x in C` to tests whether `x` is in `C`.

- Finally there is a special value called `None` to denote the absence of a value.

**Control structures**

In Python, there is no keyword for the beginning and the end of an instructions block. Blocks are delimited solely by means of indentation. Most of the time a new block is introduced by `. Python has the following control structures:

- Conditional instruction:

```
if <condition>:
    <instruction sequence>
[elif <condition>:
    <instruction sequence>]*
[else:
    <instruction sequence>]
```

- Inside expression exclusively, one can write:

```
<value> if <condition> else <value>
```

- Iterative instructions:
The `else` block is executed at the end of the loop if the loop is ended normally, that is neither by a `break` nor an exception.

- In a loop, `continue` jumps to the next iteration.
- An iterable is an object which can be iterated through. Iterable types include lists, tuples, dictionaries, and strings.
- An error (also called exception) is raised by:

```
raise <ErrorType> [, error message]
```

Usual errors include `ValueError` and `TypeError`.

**Functions**

**Note:** Python functions vs. mathematical functions

In what follows, we deal with functions is the sense of programming languages. Mathematical functions, as manipulated in calculus, are handled by Sage in a different way. In particular it doesn't make sense to do mathematical manipulation such as additions or derivations on Python functions.

One defines a function using the keyword `def` as:

```
def <name>(<argument list>):
    <instruction sequence>
```

The result of the function is given by the instruction `return`. Very short functions can be created anonymously using `lambda` (remark that there is no instruction `return` here):

```
lambda <arguments>: <expression>
```

**Note:** Functional programming

Functions are objects as any other objects. One can assign them to variables or return them. For details, see the tutorial on *Functional Programming for Mathematicians.*
Exercises

Lists

Creating Lists I: [Square brackets]

Example:

```sage
L = [3, Permutation([5, 1, 4, 2, 3]), 17, 17, 3, 51]
sage: L
[3, [5, 1, 4, 2, 3], 17, 17, 3, 51]
```

Exercise: Create the list \[63, 12, -10, "a", 12\], assign it to the variable \(L\), and print the list.

```sage
# edit here
```

Exercise: Create the empty list (you will often need to do this).

```sage
# edit here
```

Creating Lists II: range

The `range()` function provides an easy way to construct a list of integers. Here is the documentation of the `range()` function:

```
range([start[, stop[, step]]] -> list of integers

Return a list containing an arithmetic progression of integers.
range(i, j) returns \([i, i+1, i+2, \ldots, j-1]\); start (!) defaults to 0.
When step is given, it specifies the increment (or decrement). For
example, range(4) returns \([0, 1, 2, 3]\). The end point is omitted!
These are exactly the valid indices for a list of 4 elements.
```

Exercise: Use `range()` to construct the list \([1, 2, \ldots, 50]\).

```sage
# edit here
```

Exercise: Use `range()` to construct the list of even numbers between 1 and 100 (including 100).

```sage
# edit here
```

Exercise: The step argument for the `range()` command can be negative. Use range to construct the list \([10, 7, 4, 1, -2]\).

```sage
# edit here
```

See also:

- `xrange()`: returns an iterator rather than building a list, (only for Python2, replaced by range in Python 3).
- `srange()`: like range but with Sage integers; see below.
- `xsrange()`: like xrange but with Sage integers.
Creating Lists III: list comprehensions

List comprehensions provide a concise way to create lists from other lists (or other data types).

Example We already know how to create the list $[1, 2, \ldots, 16]$:

```
sage: list(range(1,17))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]
```

Using a list comprehension, we can now create the list $[1^2, 2^2, 3^2, \ldots, 16^2]$ as follows:

```
sage: [i^2 for i in range(1,17)]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256]
```

```
sage: sum([i^2 for i in range(1,17)])
1496
```

Exercise: [Project Euler, Problem 6]

The sum of the squares of the first ten natural numbers is

$$1^2 + 2^2 + \ldots + 10^2 = 385$$

The square of the sum of the first ten natural numbers is

$$(1 + 2 + \ldots + 10)^2 = 55^2 = 3025$$

Hence the difference between the sum of the squares of the first ten natural numbers and the square of the sum is

$$3025 - 385 = 2640$$

Find the difference between the sum of the squares of the first one hundred natural numbers and the square of the sum.

```
sage: # edit here
```

```
sage: # edit here
```

```
sage: # edit here
```

Filtering lists with a list comprehension

A list can be filtered using a list comprehension.

Example: To create a list of the squares of the prime numbers between 1 and 100, we use a list comprehension as follows.

```
sage: [p^2 for p in [1,2,..,100] if is_prime(p)]
[4, 9, 25, 49, 121, 169, 289, 361, 529, 841, 961, 1369, 1681, 1849, 2209, 2809, 3481, 3721, 4489, 5041, 5329, 6241, 6889, 7921, 9409]
```

Exercise: Use a list comprehension to list all the natural numbers below 20 that are multiples of 3 or 5. Hint:

- To get the remainder of 7 divided by 3 use \texttt{7\%3}.
- To test for equality use two equal signs (\texttt{==}); for example, \texttt{3 == 7}.
Project Euler, Problem 1: Find the sum of all the multiples of 3 or 5 below 1000.

Nested list comprehensions

List comprehensions can be nested!

Examples:

```
sage: [(x,y) for x in range(5) for y in range(3)]
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2), (3, 0), (3, 1),
 (3, 2), (4, 0), (4, 1), (4, 2)]
```

```
sage: [[i^j for j in range(1,4)] for i in range(6)]
[[0, 0, 0], [1, 1, 1], [2, 4, 8], [3, 9, 27], [4, 16, 64], [5, 25, 125]]
```

```
sage: matrix([[i^j for j in range(1,4)] for i in range(6)])
[ 0  0  0]
[ 1  1  1]
[ 2  4  8]
[ 3  9 27]
[ 4  16 64]
[ 5  25 125]
```

Exercise:

1. A Pythagorean triple is a triple \((x, y, z)\) of positive integers satisfying \(x^2 + y^2 = z^2\). The Pythagorean triples whose components are at most 10 are:

\[
(3, 4, 5), (4, 3, 5), (6, 8, 10), (8, 6, 10) .
\]

Using a filtered list comprehension, construct the list of Pythagorean triples whose components are at most 50:

```
sage: # edit here
```

```
sage: # edit here
```

2. Project Euler, Problem 9: There exists exactly one Pythagorean triple for which \(a + b + c = 1000\). Find the product \(abc\):

```
sage: # edit here
```
**Accessing individual elements of lists**

To access an element of the list \(L\), use the syntax \(L[i]\), where \(i\) is the index of the item.

**Exercise:**

1. Construct the list \(L = [1, 2, 3, 4, 3, 5, 6]\). What is \(L[3]\)?

   ```python
   sage: # edit here
   ```

2. What is \(L[1]\)?

   ```python
   sage: # edit here
   ```

3. What is the index of the first element of \(L\)?

   ```python
   sage: # edit here
   ```

4. What is \(L[-1]\)? What is \(L[-2]\)?

   ```python
   sage: # edit here
   ```

5. What is \(L.index(2)\)? What is \(L.index(3)\)?

   ```python
   sage: # edit here
   ```

**Modifying lists: changing an element in a list**

To change the item in position \(i\) of a list \(L\):

```python
sage: L = ["a", 4, 1, 8]
sage: L
['a', 4, 1, 8]

sage: L[2] = 0
sage: L
['a', 4, 0, 8]
```

**Modifying lists: append and extend**

To *append* an object to a list:

```python
sage: L = ["a", 4, 1, 8]
sage: L
['a', 4, 1, 8]

sage: L.append(17)
sage: L
['a', 4, 1, 8, 17]
```

To *extend* a list by another list:
sage: L1 = [1,2,3]
sage: L2 = [7,8,9,0]

sage: L1
[1, 2, 3]
sage: L2
[7, 8, 9, 0]

sage: L1.extend(L2)
sage: L1
[1, 2, 3, 7, 8, 9, 0]

Modifying lists: reverse, sort, ...

sage: L = [4,2,5,1,3]
sage: L
[4, 2, 5, 1, 3]

sage: L.reverse()
sage: L
[3, 1, 5, 2, 4]

sage: L.sort()
sage: L
[1, 2, 3, 4, 5]

sage: L = [3,1,6,4]
sage: sorted(L)
[1, 3, 4, 6]

sage: L
[3, 1, 6, 4]

Concatenating Lists

To concatenate two lists, add them with the operator +. This is not a commutative operation!

sage: L1 = [1,2,3]
sage: L2 = [7,8,9,0]
sage: L1 + L2
[1, 2, 3, 7, 8, 9, 0]
Slicing Lists

You can slice a list using the syntax `L[start : stop : step]`. This will return a sublist of `L`.

**Exercise:** Below are some examples of slicing lists. Try to guess what the output will be before evaluating the cell:

```python
sage: L = list(range(20))
sage: L
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

```python
sage: L[3:15]
[3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

```python
sage: L[3:15:2]
[3, 5, 7, 9, 11, 13]
```

```python
sage: L[15:3:-1]
[15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4]
```

```python
sage: L[:4]
[0, 1, 2, 3]
```

```python
sage: L[:]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

```python
sage: L[::-1]
[19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

**Exercise (Advanced):** The following function combines a loop with some of the list operations above. What does the function do?

```python
sage: def f(number_of_iterations):
    ....:     L = [1]
    ....:     for n in range(2, number_of_iterations):
    ....:         L = [sum(L[:i]) for i in range(n-1, -1, -1)]
    ....:     return numerical_approx(2*L[0]*len(L)/sum(L), digits=50)

sage: # edit here
```

Tuples

A tuple is an immutable list. That is, it cannot be changed once it is created. This can be useful for code safety and foremost because it makes tuple hashable. To create a tuple, use parentheses instead of brackets:

```python
sage: t = (3, 5, [3, 1], (17, [2, 3], 17), 4)
sage: t
(3, 5, [3, 1], (17, [2, 3], 17), 4)
```

To create a singleton tuple, a comma is required to resolve the ambiguity:
We can create a tuple from a list, and vice-versa.

```python
sage: tuple(range(5))
(0, 1, 2, 3, 4)
```

```python
sage: list(t)
[3, 5, [3, 1], (17, [2, 3], 17), 4]
```

Tuples behave like lists in many respects:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Syntax for lists</th>
<th>Syntax for tuples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concatenation</td>
<td>list1 + list2</td>
<td>tuple1 + tuple2</td>
</tr>
<tr>
<td>A reversed copy</td>
<td>list[::-1]</td>
<td>tuple[::-1]</td>
</tr>
<tr>
<td>Length</td>
<td>len(list)</td>
<td>len(tuple)</td>
</tr>
</tbody>
</table>

Trying to modify a tuple will fail:

```python
sage: t = (5, 'a', 6/5)
sage: t
(5, 'a', 6/5)
```

```python
sage: t[1] = 'b'
Traceback (most recent call last):
...
TypeError: 'tuple' object does not support item assignment
```

**Generators**

"Tuple-comprehensions" do not exist. Instead, the syntax produces something called a generator. A generator allows you to process a sequence of items one at a time. Each item is created when it is needed, and then forgotten. This can be very efficient if we only need to use each item once.

```python
sage: (i^2 for i in range(5))
<generator object <genexpr> at 0x...>
```

```python
sage: g = (i^2 for i in range(5))
sage: g[0]
Traceback (most recent call last):
...
TypeError: 'generator' object ...
```

```python
sage: [x for x in g]
[0, 1, 4, 9, 16]
```
g is now empty.

```
sage: [x for x in g]
[]
```

A nice ‘pythonic’ trick is to use generators as argument of functions. We do not need double parentheses for this:

```
sage: sum( i^2 for i in srange(100001) )
333338333350000
```

### Dictionaries

A dictionary is another built-in data type. Unlike lists, which are indexed by a range of numbers starting at 0, dictionaries are indexed by keys, which can be any immutable objects. Strings and numbers can always be keys (because they are immutable). Dictionaries are sometimes called “associative arrays” in other programming languages.

There are several ways to define dictionaries. One method is to use braces, {}, with comma-separated entries given in the form key:value:

```
sage: d = {3:17, 0.5:[4,1,5,2,3], 0:“goo”, 3/2 : 17}
sage: d
{0: 'goo', 0.500000000000000: [4, 1, 5, 2, 3], 3/2: 17, 3: 17}
```

A second method is to use the constructor `dict` which admits a list (or actually any iterable) of 2-tuples (key, value):

```
sage: dd = dict((i,i^2) for i in range(10))
sage: dd
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

Dictionaries behave as lists and tuples for several important operations.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Syntax for lists</th>
<th>Syntax for dictionaries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accessing elements</td>
<td>list[3]</td>
<td>D[“key”]</td>
</tr>
<tr>
<td>Length</td>
<td>len(list)</td>
<td>len(D)</td>
</tr>
<tr>
<td>Modifying</td>
<td>L[3] = 17</td>
<td>D[“key”] = 17</td>
</tr>
<tr>
<td>Deleting items</td>
<td>del L[3]</td>
<td>del D[“key”]</td>
</tr>
</tbody>
</table>

```
sage: d[10]='a'
sage: d
{0: 'goo', 0.500000000000000: [4, 1, 5, 2, 3], 3/2: 17, 3: 17, 10: 'a'}
```

A dictionary can have the same value multiple times, but each key must only appear once and must be immutable:

```
sage: d = {3: 14, 4: 14}
sage: d
{3: 14, 4: 14}
```

```
sage: d = {3: 13, 3: 14}
sage: d
{3: 14}
```
Another way to add items to a dictionary is with the `update()` method which updates the dictionary from another dictionary:

```
sage: d = {}
sage: d
{}

sage: d.update({10 : 'newvalue', 20: 'newervalue', 3: 14, 0.5:[1,2,3]})
sage: d
{0.500000000000000: [1, 2, 3], 3: 14, 10: 'newvalue', 20: 'newervalue'}
```

We can iterate through the keys, or values, or both, of a dictionary. Note that, internally, there is no sorting of keys done. In general, the order of keys/values will depend on memory locations can and will differ between different computers and / or repeated runs on the same computer. However, Sage sort the dictionary entries by key when printing the dictionary specifically to make the docstrings more reproducible. However, the Python methods `keys()` and `values()` do not sort for you. If you want your output to be reproducible, then you have to sort it first just like in the examples below:

```
sage: d = {10 : 'newvalue', 20: 'newervalue', 3: 14, 0.5:[1,2,3]}

sage: sorted([key for key in d])
[0.500000000000000, 3, 10, 20]

sage: d.keys()  # random order
[0.500000000000000, 10, 3, 20]
sage: sorted(d.keys())
[0.500000000000000, 3, 10, 20]

sage: d.values()  # random order
[(1, 2, 3), 'newvalue', 14, 'newervalue']
sage: set(d.values()) == set([14, (1, 2, 3), 'newvalue', 'newervalue'])
True

sage: d.items()  # random order
[(0.500000000000000, (1, 2, 3)), (10, 'newvalue'), (3, 14), (20, 'newervalue')]
sage: sorted([(key, value) for key, value in d.items()])
[(0.500000000000000, (1, 2, 3)), (3, 14), (10, 'newvalue'), (20, 'newervalue')]
```

**Exercise:** Consider the following directed graph.
Create a dictionary whose keys are the vertices of the above directed graph, and whose values are the lists of the vertices that it points to. For instance, the vertex 1 points to the vertices 2 and 3, so the dictionary will look like:

\[
d = \{ \ldots, 1: [2, 3], \ldots \}
\]

`sage: # edit here`

Then try:

```sage
sage: g = DiGraph(d)
sage: g.plot()
```

**Using Sage types: The `srange` command**

**Example:** Construct a $3 \times 3$ matrix whose $(i, j)$ entry is the rational number $\frac{i}{j}$. The integers generated by `range()` are Python `int`'s. As a consequence, dividing them does euclidean division (in Python2):

```sage
sage: matrix([[i/j for j in range(1,4)] for i in range(1,4)])  # not tested
[1 0 0]
[2 1 0]
[3 1 1]
```

In Python3, the division of Python integers returns a float instead.

Whereas dividing a Sage `Integer` by a Sage `Integer` produces a rational number:

```sage
sage: matrix([[ i/j for j in srange(1,4)] for i in srange(1,4)])
[ 1 1/2 1/3]
[ 2 1 2/3]
[ 3 3/2 1]
```
Modifying lists has consequences!

Try to predict the results of the following commands:

```plaintext
sage: a = [1, 2, 3]
sage: L = [a, a, a]
sage: L
[[1, 2, 3], [1, 2, 3], [1, 2, 3]]
```

```plaintext
sage: a.append(4)
sage: L
[[1, 2, 3, 4], [1, 2, 3, 4], [1, 2, 3, 4]]
```

Now try these:

```plaintext
sage: a = [1, 2, 3]
sage: L = [a, a, a]
sage: L
[[1, 2, 3], [1, 2, 3], [1, 2, 3]]
```

```plaintext
sage: a = [1, 2, 3, 4]
sage: L
[[1, 2, 3], [1, 2, 3], [1, 2, 3]]
```

```plaintext
sage: L[0].append(4)
sage: L
[[1, 2, 3, 4], [1, 2, 3, 4], [1, 2, 3, 4]]
```

This is known as the reference effect. You can use the command `deepcopy()` to avoid this effect:

```plaintext
sage: a = [1, 2, 3]
sage: L = [deepcopy(a), deepcopy(a)]
sage: L
[[1, 2, 3], [1, 2, 3]]
```

```plaintext
sage: a.append(4)
sage: L
[[1, 2, 3], [1, 2, 3]]
```

The same effect occurs with dictionaries:

```plaintext
sage: d = {1: 'a', 2: 'b', 3: 'c'}
sage: dd = d
sage: d.update( { 4: 'd' } )
sage: dd
{1: 'a', 2: 'b', 3: 'c', 4: 'd'}
```
Loops and Functions

For more verbose explanation of what’s going on here, a good place to look at is the following section of the Python tutorial: http://docs.python.org/tutorial/controlflow.html

While Loops

While loops tend not to be used nearly as much as for loops in Python code:

```python
sage: i = 0
sage: while i < 10:
    ....:     print(i)
    ....:     i += 1
0
1
2
3
4
5
6
7
8

sage: i = 0
sage: while i < 10:
    ....:     if i % 2 == 1:
    ....:         i += 1
    ....:         continue
    ....:     print(i)
    ....:     i += 1
0
2
4
6
8
```

Note that the truth value of the clause expression in the while loop is evaluated using bool:

```python
sage: bool(True)
True

sage: bool('a')
True

sage: bool(1)
True

sage: bool(0)
False
```
```python
sage: i = 4
sage: while i:
    ....:     print(i)
    ....:     i -= 1
4
3
2
1
```

### For Loops

Here is a basic `for` loop iterating over all of the elements in the list `l`:

```python
sage: l = ['a', 'b', 'c']
sage: for letter in l:
    ....:     print(letter)
a
b
c
```

The `range()` function is very useful when you want to generate arithmetic progressions to loop over. Note that the end point is never included:

```python
sage: range
```

```python
sage: list(range(4))
[0, 1, 2, 3]
```

```python
sage: list(range(1, 5))
[1, 2, 3, 4]
```

```python
sage: list(range(1, 11, 2))
[1, 3, 5, 7, 9]
```

```python
sage: list(range(10, 0, -1))
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

```python
sage: for i in range(4):
    ....:     print("{} {}".format(i, i*i))
0
1
2
3
```

You can use the `continue` keyword to immediately go to the next item in the loop:

```python
sage: for i in range(10):
    ....:     if i % 2 == 0:
    ....:         continue
    ....:     print(i)
(continues on next page)
```
If you want to break out of the loop, use the *break* keyword:

```python
sage: for i in range(10):
    ....:     if i % 2 == 0:
    ....:         continue
    ....:     if i == 7:
    ....:         break
    ....:     print(i)
1
3
5
7
9
```

If you need to keep track of both the position in the list and its value, one (not so elegant) way would be to do the following:

```python
sage: l = ['a', 'b', 'c']
sage: for i in range(len(l)):
    ....:     print("{} {}".format(i, l[i]))
0 a
1 b
2 c
```

It's cleaner to use *enumerate()* which provides the index as well as the value:

```python
sage: l = ['a', 'b', 'c']
sage: for i, letter in enumerate(l):
    ....:     print("{} {}".format(i, letter))
0 a
1 b
2 c
```

You could get a similar result to the result of the *enumerate()* function by using *zip()* to zip two lists together:

```python
sage: l = ['a', 'b', 'c']
sage: for i, letter in zip(range(len(l)), l):
    ....:     print("{} {}".format(i, letter))
0 a
1 b
2 c
```

*For* loops work using Python's iterator protocol. This allows all sorts of different objects to be looped over. For example:

```python
sage: for i in GF(5):
    ....:     print("{} {}".format(i, i*i))
0 0
1 1
2 4
```

(continues on next page)
How does this work?

```
sage: it = iter(GF(5)); it
<generator object ...__iter__ at 0x...>

sage: next(it)
0

sage: next(it)
1

sage: next(it)
2

sage: next(it)
3

sage: next(it)
4
```

```
sage: R = GF(5)
sage: R.__iter__??
```

The command `yield` provides a very convenient way to produce iterators. We’ll see more about it in a bit.

**Exercises**

For each of the following sets, compute the list of its elements and their sum. Use two different ways, if possible: with a loop, and using a list comprehension.

1. The first $n$ terms of the harmonic series:

   \[ \sum_{i=1}^{n} \frac{1}{i} \]

   ```
   sage: # edit here
   ```

2. The odd integers between 1 and $n$:

   ```
   sage: # edit here
   ```

3. The first $n$ odd positive integers:
Functions

Functions are defined using the `def` statement, and values are returned using the `return` keyword:

```python
sage: def f(x):
....:     return x**x
sage: f(2)
4
```

Functions can be recursive:

```python
sage: def fib(n):
....:     if n <= 1:
....:         return 1
....:     else:
....:         return fib(n-1) + fib(n-2)
sage: [fib(i) for i in range(10)]
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

Functions are first class objects like any other. For example, they can be passed in as arguments to other functions:

```python
sage: def compose(f, x, n):
# computes f(f(...f(x)))
....:     for i in range(n):
....:         x = f(x)  # this change is local to this function call!
....:     return x
sage: compose(f, 2, 3)
256
sage: def add_one(x):
....:     return x + 1
sage: compose(add_one, 2, 3)
5
```

You can give default values for arguments in functions:
You can return multiple values from a function:

```python
sage: def g(x):
    ...:     return x, x**x
```
```
sage: g(2)
(2, 4)
```
```
sage: type(g)
<... 'function'>
```
```
sage: a, b = g(100)
```
```
sage: a
100
```
```
sage: b
10000
```

You can also take a variable number of arguments and keyword arguments in a function:

```python
sage: def h(*args, **kwds):
    ...:     print("{} {}".format(type(args), args))
    ...:     print("{} {}".format(type(kwds), kwds))
```
```
sage: h(1,2,3,n=4)
<... 'tuple'> (1, 2, 3)
<... 'dict'> {'n': 4}
```

Let's use the `yield` instruction to make a generator for the Fibonacci numbers up to $n$:

```python
sage: def fib_gen(n):
    ...:     if n < 1:
    ...:         return
    ...:     a = b = 1
    ...:     yield b
    ...:     while b < n:
    ...:         yield b
    ...:         a, b = b, b+a
```
```
```
sage: for i in fib_gen(50):
....:    print(i)
1
1
2
3
5
8
13
21
34

Exercises

1. Write a function is_even which returns True if \( n \) is even and False otherwise.
2. Write a function every_other which takes a list \( l \) as input and returns a list containing every other element of \( l \).
3. Write a generator every_other which takes an iterable \( l \) as input, and returns every other element of \( l \), one after the other.
4. Write a function which computes the \( n \)-th Fibonacci number. Try to improve performance.

Todo:

- Definition of hashable
- Introduction to the debugger.

12.1.13 Tutorial: Comprehensions, Iterators, and Iterables

Author: Florent Hivert <florent.hivert@univ-rouen.fr> and Nicolas M. Thiéry <nthiery at users.sf.net>

List comprehensions

List comprehensions are a very handy way to construct lists in Python. You can use either of the following idioms:

\[
[ \text{<expr>} \ \text{for} \ \text{<name>} \ \text{in} \ \text{<iterable>} ] \\
[ \text{<expr>} \ \text{for} \ \text{<name>} \ \text{in} \ \text{<iterable>} \ \text{if} \ \text{<condition>} ]
\]

For example, here are some lists of squares:

sage: [ i^2 for i in [1, 3, 7] ]
[1, 9, 49]
sage: [ i^2 for i in range(1,10) ]
[1, 4, 9, 16, 25, 36, 49, 64, 81]
sage: [ i^2 for i in range(1,10) if i % 2 == 1]
[1, 9, 25, 49, 81]

And a variant on the latter:
Exercises

1. Construct the list of the squares of the prime integers between 1 and 10:

```python
sage: # edit here
```

2. Construct the list of the perfect squares less than 100 (hint: use `srange()` to get a list of Sage integers together with the method `i.sqrtrem()`):

```python
sage: # edit here
```

One can use more than one iterable in a list comprehension:

```python
sage: [(i,j) for i in range(1,6) for j in range(1,i)]
[(2, 1), (3, 1), (3, 2), (4, 1), (4, 2), (4, 3), (5, 1), (5, 2), (5, 3), (5, 4)]
```

**Warning**: Mind the order of the nested loop in the previous expression.

If instead one wants to build a list of lists, one can use nested lists as in:

```python
sage: [[ binomial(n, i) for i in range(n+1) ] for n in range(10) ]
[[1],
 [1, 1],
 [1, 2, 1],
 [1, 3, 3, 1],
 [1, 4, 6, 4, 1],
 [1, 5, 10, 10, 5, 1],
 [1, 6, 15, 20, 15, 6, 1],
 [1, 7, 21, 35, 35, 21, 7, 1],
 [1, 8, 28, 56, 70, 56, 28, 8, 1],
 [1, 9, 36, 84, 126, 126, 84, 36, 9, 1]]
```

Exercises

1. Compute the list of pairs \((i, j)\) of non negative integers such that \(i\) is at most 5, \(j\) is at most 8, and \(i\) and \(j\) are co-prime:

```python
sage: # edit here
```

2. Compute the same list for \(i < j < 10\):

```python
sage: # edit here
```
Iterators

Definition

To build a comprehension, Python actually uses an iterator. This is a device which runs through a bunch of objects, returning one at each call to the next method. Iterators are built using parentheses:

```python
sage: it = (binomial(8, i) for i in range(9))
sage: next(it)
1
sage: next(it)
8
sage: next(it)
28
sage: next(it)
56
```

You can get the list of the results that are not yet consumed:

```python
sage: list(it)
[70, 56, 28, 8, 1]
```

Asking for more elements triggers a StopIteration exception:

```python
sage: next(it)
Traceback (most recent call last):
  ...
StopIteration
```

An iterator can be used as argument for a function. The two following idioms give the same results; however, the second idiom is much more memory efficient (for large examples) as it does not expand any list in memory:

```python
sage: sum([binomial(8, i) for i in range(9)])
256
sage: sum(binomial(8, i) for i in range(9))
256
```

Exercises

1. Compute the sum of \( \binom{10}{i} \) for all even \( i \):

   ```python
   sage: # edit here
   ```

2. Compute the sum of the products of all pairs of co-prime numbers \( i, j \) for \( i < j < 10 \):

   ```python
   sage: # edit here
   ```
Typical usage of iterators

Iterators are very handy with the functions `all()`, `any()`, and `exists()`:

```python
sage: all([True, True, True, True])
True
sage: all([True, False, True, True])
False

sage: any([False, False, False, False])
False
sage: any([False, False, True, False])
True
```

Let's check that all the prime numbers larger than 2 are odd:

```python
sage: all( is_odd(p) for p in range(1,100) if is_prime(p) and p>2 )
True
```

It is well know that if $2^p-1$ is prime then $p$ is prime:

```python
sage: def mersenne(p):
    return 2^p -1
sage: [ is_prime(p) for p in range(20) if is_prime(mersenne(p)) ]
[True, True, True, True, True, True, True]
```

The converse is not true:

```python
sage: all( is_prime(mersenne(p)) for p in range(1000) if is_prime(p) )
False
```

Using a list would be much slower here:

```python
sage: %time all( is_prime(mersenne(p)) for p in range(1000) if is_prime(p) )  # not...tested
CPU times: user 0.00 s, sys: 0.00 s, total: 0.00 s
Wall time: 0.00 s
False

sage: %time all( [ is_prime(mersenne(p)) for p in range(1000) if is_prime(p)] )  # not...tested
CPU times: user 0.72 s, sys: 0.00 s, total: 0.73 s
Wall time: 0.73 s
False
```

You can get the counterexample using `exists()`. It takes two arguments: an iterator and a function which tests the property that should hold:

```python
sage: exists( (p for p in range(1000) if is_prime(p)), lambda p: not is_prime(mersenne(p)) )
(True, 11)
```

An alternative way to achieve this is:
sage: counter_examples = (p for p in range(1000) if is_prime(p) and not is_prime(mersenne(p)))
sage: next(counter_examples)
11

Exercises

1. Build the list \{i^3 \mid -10 < i < 10\}. Can you find two of those cubes \(u\) and \(v\) such that \(u + v = 218\)?

sage: # edit here

**itertools**

At its name suggests **itertools** is a module which defines several handy tools for manipulating iterators:

sage: l = [3, 234, 12, 53, 23]
sage: [(i, l[i]) for i in range(len(l))]
[(0, 3), (1, 234), (2, 12), (3, 53), (4, 23)]

The same results can be obtained using **enumerate()**:

sage: list(enumerate(l))
[(0, 3), (1, 234), (2, 12), (3, 53), (4, 23)]

Here is the analogue of list slicing:

sage: list(Permutations(3))
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
sage: list(Permutations(3))[1:4]
[[1, 3, 2], [2, 1, 3], [2, 3, 1]]

sage: import itertools
sage: list(itertools.islice(Permutations(3), 1r, 4r))
[[1, 3, 2], [2, 1, 3], [2, 3, 1]]

Note that all calls to islice must have arguments of type int and not Sage integers.

The behaviour of the functions **map()** and **filter()** has changed between Python 2 and Python 3. In Python 3, they return an iterator. If you want to return a list like in Python 2 you need to explicitly wrap them in **list()**:

sage: list(map(lambda z: z.cycle_type(), Permutations(3)))
[[1, 1, 1], [2, 1], [2, 1], [3], [3], [2, 1]]
sage: list(filter(lambda z: z.has_pattern([1,2]), Permutations(3)))
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2]]

Exercises

1. Define an iterator for the \(i\)-th prime for \(5 < i < 10\):

sage: # edit here
Defining new iterators

One can very easily write new iterators using the keyword yield. The following function does nothing interesting beyond demonstrating the use of yield:

```python
sage: def f(n):
    ....:     for i in range(n):
    ....:         yield i
sage: [ u for u in f(5) ]
[0, 1, 2, 3, 4]
```

Iterators can be recursive:

```python
sage: def words(alphabet,l):
    ....:     if l == 0:
    ....:         yield []
    ....:     else:
    ....:         for word in words(alphabet, l-1):
    ....:             for a in alphabet:
    ....:                 yield word + [a]

sage: [ w for w in words([['a', 'b', 'c'], 3]) ]
[['a', 'a'], ['a', 'a', 'a'], ['a', 'a', 'b'], ['a', 'a', 'c'], ['a', 'b', 'a'], ['a', 'b', 'b'], ['a', 'b', 'c'], ['a', 'c', 'a'], ['a', 'c', 'b'], ['a', 'c', 'c'], ['b', 'a', 'a'], ['b', 'a', 'b'], ['b', 'a', 'c'], ['b', 'b', 'a'], ['b', 'b', 'b'], ['b', 'b', 'c'], ['b', 'c', 'a'], ['b', 'c', 'b'], ['b', 'c', 'c'], ['c', 'a', 'a'], ['c', 'a', 'b'], ['c', 'a', 'c'], ['c', 'b', 'a'], ['c', 'b', 'b'], ['c', 'b', 'c'], ['c', 'c', 'a'], ['c', 'c', 'b'], ['c', 'c', 'c']]

sage: sum(1 for w in words([['a', 'b', 'c'], 3]))
27
```

Here is another recursive iterator:

```python
sage: def dyck_words(l):
    ....:     if l==0:
    ....:         yield ''
    ....:     else:
    ....:         for k in range(l):
    ....:             for w1 in dyck_words(k):
    ....:                 for w2 in dyck_words(l-k-1):
    ....:                     yield '('+w1+')'+w2

sage: list(dyck_words(4))
['(())()()', '(())(()())', '(())(())()', '(())(())()', '(()())()()', '(()())()()', '(()())()()', '(())()()()', '(())()()()', '(())()()()', '(())()()()', '(())()()()', '(())()()()', '(())()()()', '(())()()()', '(())()()()', '(())()()()', '(())()()()', '(())()()()', '(())()()()', '(())()()()', '(())()()()', '(())()()()', '(())()()()', '(())()()()', '(())()()()', '(())()()()', '(())()()()', '(())()()()', '(())()()()', '(())()()()', '(())()()()', '(())()()()', '(())()()()', '(())()()()', '(())()()()', '(())()()()', '(())()()()', '(())()()()', '(())()()()', '(())()()()', '(())()()()', '(())()()()', '(())()()()', '(())()()()', '(())()()()', '(())()()()', '(())()()()', '(())()()()', '(())()()()', '(())()()()', '(())()()()', '(())()()()', '(())()()()', '(())()()()', (continues on next page)
Exercises

1. Write an iterator with two parameters $n, l$ iterating through the set of nondecreasing lists of integers smaller than $n$ of length $l$:

```python
sage: # edit here
```

Standard Iterables

Finally, many standard Python and Sage objects are *iterable*; that is one may iterate through their elements:

```python
sage: sum( x^len(s) for s in Subsets(8) )
x^8 + 8*x^7 + 28*x^6 + 56*x^5 + 70*x^4 + 56*x^3 + 28*x^2 + 8*x + 1

sage: sum( x^p.length() for p in Permutations(3) )
x^3 + 2*x^2 + 2*x + 1

sage: factor(sum( x^p.length() for p in Permutations(3) ))
(x^2 + x + 1)*(x + 1)

sage: P = Permutations(5)
sage: all( p in P for p in P )
True

sage: for p in GL(2, 2): print(p); print('')
[1 0]
[0 1]

[0 1]
[1 0]

[0 1]
[1 1]

[1 1]
[0 1]

[1 1]
[1 0]
```
sage: for p in Partitions(3): print(p)
[3]
[2, 1]
[1, 1, 1]

Beware of infinite loops:

sage: for p in Partitions(): print(p) # not tested
sage: for p in Primes(): print(p)    # not tested

Infinite loops can nevertheless be very useful:

sage: exists( Primes(), lambda p: not is_prime(mersenne(p)) )
(True, 11)

sage: counter_examples = (p for p in Primes() if not is_prime(mersenne(p)))
sage: next(counter_examples)
11

12.1.14 Tutorial: Objects and Classes in Python and Sage

Author: Florent Hivert <florent.hivert@univ-rouen.fr>

This tutorial is an introduction to object-oriented programming in Python and Sage. It requires basic knowledge about imperative/procedural programming (the most common programming style) – that is, conditional instructions, loops, functions (see the “Programming” section of the Sage tutorial) – but no further knowledge about objects and classes is assumed. It is designed as an alternating sequence of formal introduction and exercises. Solutions to the exercises are given at the end.

Foreword: variables, names and objects

As an object-oriented language, Python’s “variables” behavior may be surprising for people used to imperative languages like C or Maple. The reason is that they are not variables but names.

The following explanation is borrowed from David Goodger.
Other languages have “variables”

In many other languages, assigning to a variable puts a value into a box.

```
int a = 1;
```

Box “a” now contains an integer 1.

Assigning another value to the same variable replaces the contents of the box:

```
a = 2;
```

Now box “a” contains an integer 2.

Assigning one variable to another makes a copy of the value and puts it in the new box:

```
int b = a;
```
“b” is a second box, with a copy of integer 2. Box “a” has a separate copy.

**Python has “names”**

In Python, a “name” or “identifier” is like a parcel tag (or nametag) attached to an object.

```
1
a = 1
```

Here, an integer 1 object has a tag labelled “a”.

If we reassign to “a”, we just move the tag to another object:

```
2
a = 2
```

Now the name “a” is attached to an integer 2 object.

The original integer 1 object no longer has a tag “a”. It may live on, but we can’t get to it through the name “a”. (When an object has no more references or tags, it is removed from memory.)

If we assign one name to another, we’re just attaching another nametag to an existing object:

```
2
b = a
```

The name “b” is just a second tag bound to the same object as “a”.

Although we commonly refer to “variables” even in Python (because it’s common terminology), we really mean “names” or “identifiers”. In Python, “variables” are nametags for values, not labelled boxes.
Warning: As a consequence, when there are two tags “a” and “b” on the same object, modifying the object tagged “b” also modifies the object tagged “a”:

```sage
a = [1,2,3]
sage: b = a
sage: b[1] = 0
sage: a
[1, 0, 3]
```

Note that reassigning the tag “b” (rather than modifying the object with that tag) doesn’t affect the object tagged “a”:

```sage
b = 7
sage: b
7
sage: a
[1, 0, 3]
```

Object-oriented programming paradigm

The object-oriented programming paradigm relies on the two following fundamental rules:

1. Anything of the real (or mathematical) world which needs to be manipulated by the computer is modeled by an object.
2. Each object is an instance of some class.

At this point, those two rules are a little meaningless, so let’s give some more or less precise definitions of the terms:

**object** a portion of memory which contains the information needed to model the real world thing.

**class** defines the data structure used to store the objects which are instances of the class together with their behavior.

Let’s start with some examples: We consider the vector space over $\mathbb{Q}$ whose basis is indexed by permutations, and a particular element in it:

```sage
F = CombinatorialFreeModule(QQ, Permutations())
sage: el = 3*F([1,3,2]) + F([1,2,3])
sage: el
B[[1, 2, 3]] + 3*B[[1, 3, 2]]
```

(For each permutation, say [1, 3, 2], the corresponding element in F is denoted by $B[[1, 3, 2]]$ – in a CombinatorialFreeModule, if an element is indexed by $x$, then by default its print representation is $B[x]$.)

In Python, everything is an object so there isn’t any difference between types and classes. One can get the class of the object `el` by:

```sage
type(el)
<class 'sage.combinat.free_module.CombinatorialFreeModule_with_category.element_class'>
```

As such, this is not very informative. We’ll come back to it later. The data associated to objects are stored in so-called attributes. They are accessed through the syntax `obj.attribute_name`. For an element of a combinatorial free module, the main attribute is called `_monomial_coefficients`. It is a dictionary associating coefficients to indices:
Thematic Tutorials, Release 9.7

```python
sage: el._monomial_coefficients
{[1, 2, 3]: 1, [1, 3, 2]: 3}
```

Modifying the attribute modifies the objects:

```python
sage: el._monomial_coefficients[Permutation([3,2,1])] = 1/2
sage: el
B[[1, 2, 3]] + 3*B[[1, 3, 2]] + 1/2*B[[3, 2, 1]]
```

**Warning:** as a user, you are not supposed to do such a modification by yourself (see note on private attributes below).

As an element of a vector space, `el` has a particular behavior:

```python
sage: 2*el
2*B[[1, 2, 3]] + 6*B[[1, 3, 2]] + B[[3, 2, 1]]
sage: sorted(el.support())
[[1, 2, 3], [1, 3, 2], [3, 2, 1]]
sage: el.coefficient([1, 2, 3])
1
```

The behavior is defined through methods (`support`, `coefficient`). Note that this is true even for equality, printing or mathematical operations. For example, the call `a == b` actually is translated to the method call `a.__eq__(b)`. The names of those special methods which are usually called through operators are fixed by the Python language and are of the form `__name__`. Examples include `__eq__` and `__le__` for operators `==` and `<=`, `__repr__` (see Sage specifics about classes) for printing, `__add__` and `__mul__` for operators `+` and `*`. See [http://docs.python.org/library/](http://docs.python.org/library/) for a complete list.

```python
sage: el.__eq__(F([1,3,2]))
False
e sage: el.__repr__()
'B[[1, 2, 3]] + 3*B[[1, 3, 2]] + 1/2*B[[3, 2, 1]]'
```

**Note:** We now create a custom `Element` class to explain the details of how attributes work in Python (you can ignore the parent in the code below, that is not relevant here):

```python
sage: from sage.structure.element import Element
sage: class MyElt(Element):
....:     def __init__(self, parent, val):
....:         super(MyElt, self).__init__(parent)
....:         self.value = val
sage: el = MyElt(val=42, parent=ZZ)
e sage: el
Generic element of a structure
```

The class is stored in a particular attribute called `__class__`, and the normal attributes are stored in a dictionary called `__dict__`:
Some particular actions modify the attributes of `el`:

```python
sage: el.rename("bla")
sage: el
bla
sage: el.__dict__
{'__custom_name': 'bla', 'value': 42}
```

Lots of Sage objects are not Python objects but compiled Cython objects. Python sees them as builtin objects and you do not have access to some of their internal data structure. For example, the base class `Element` stores the parent of `el` as a Cython attribute `_parent` but it does not appear in the `__dict__` and we cannot access it from Python.

Some examples of Cython classes (technically, *extension types*) in Sage include integers and permutation group elements. These do not have a `__dict__` at all:

```python
sage: e = Integer(9)
sage: type(e)
<class 'sage.rings.integer.Integer'>
sage: e.__dict__
Traceback (most recent call last):
  ...
AttributeError: 'sage.rings.integer.Integer' object has no attribute '__dict__'

sage: id4 = SymmetricGroup(4).one()
sage: type(id4)
<class 'sage.groups.perm_gps.permgroup_element.SymmetricGroupElement'>
sage: id4.__dict__
Traceback (most recent call last):
  ...
AttributeError: 'sage.groups.perm_gps.permgroup_element.SymmetricGroupElement' object has no attribute '__dict__'
```

**Note:** Each object corresponds to a portion of memory called its *identity* in Python. You can get the identity using `id`:

```python
sage: el = Integer(9)
sage: id(el)  # random
139813642977744
sage: el1 = el; id(el1) == id(el)
True
sage: el1 is el
True
```

In Python (and therefore in Sage), two objects with the same identity will be equal, but the converse is not true in general. Thus the identity function is different from mathematical identity:
Summary

To define some object, you first have to write a class. The class will define the methods and the attributes of the object.

**method** particular kind of function associated with an object used to get information about the object or to manipulate it.

**attribute** variable where information about the object is stored.

An example: glass of beverage in a restaurant

Let’s write a small class about glasses in a restaurant:

```python
sage: class Glass:
    ....:     def __init__(self, size):
    ....:         assert size > 0
    ....:         self._size = float(size)  # an attribute
    ....:         self._content = float(0.0)  # another attribute
    ....:     def __repr__(self):
    ....:         if self._content == 0.0:
    ....:             return "An empty glass of size %s"%(self._size)
    ....:         else:
    ....:             return "A glass of size %s cl containing %s cl of water"%(self._size, self._content)
    ....:     def fill(self):
    ....:         self._content = self._size
    ....:     def empty(self):
    ....:         self._content = float(0.0)
```

Let’s create a small glass:

```python
sage: myGlass = Glass(10); myGlass
An empty glass of size 10.0
sage: myGlass.fill(); myGlass
A glass of size 10.0 cl containing 10.0 cl of water
sage: myGlass.empty(); myGlass
An empty glass of size 10.0
```

Some comments:

1. The definition of the class Glass defines two attributes, _size and _content. It defines four methods, __init__, __repr__, fill, and empty. (Any instance of this class will also have other attributes and methods, inherited from the class object. See Inheritance below.)
2. The method \_\_init\_\_ is used to initialize the object: it is used by the so-called constructor of the class that is executed when calling Glass(10).

3. The method \_\_repr\_\_ returns a string which is used to print the object, for example in this case when evaluating myGlass.

**Note: Private Attributes**

- Most of the time, in order to ensure consistency of the data structures, the user is not supposed to directly change certain attributes of an object. Those attributes are called private. Since there is no mechanism to ensure privacy in Python, the convention is the following: private attributes have names beginning with an underscore.

- As a consequence, attribute access is only made through methods. Methods for reading or writing a private attribute are called accessors.

- Methods which are only for internal use are also prefixed with an underscore.

**Exercises**

1. Add a method is_empty which returns true if a glass is empty.

2. Define a method drink with a parameter amount which allows one to partially drink the water in the glass. Raise an error if one asks to drink more water than there is in the glass or a negative amount of water.

3. Allows the glass to be filled with wine, beer or another beverage. The method fill should accept a parameter beverage. The beverage is stored in an attribute _beverage. Update the method \_\_repr\_\_ accordingly.

4. Add an attribute _clean and methods is_clean and wash. At the creation a glass is clean, as soon as it’s filled it becomes dirty, and it must be washed to become clean again.

5. Test everything.

6. Make sure that everything is tested.

7. Test everything again.

**Inheritance**

The problem: objects of different classes may share a common behavior. For example, if one wants to deal with different dishes (forks, spoons, …), then there is common behavior (becoming dirty and being washed). So the different classes associated to the different kinds of dishes should have the same clean, is_clean and wash methods. But copying and pasting code is very bad for maintenance: mistakes are copied, and to change anything one has to remember the location of all the copies. So there is a need for a mechanism which allows the programmer to factorize the common behavior. It is called inheritance or sub-classing: one writes a base class which factorizes the common behavior and then reuses the methods from this class.

We first write a small class “AbstractDish” which implements the “clean-dirty-wash” behavior:

```
sage: class AbstractDish:
....:     def __init__(self):
....:         self._clean = True
....:     def is_clean(self):
....:         return self._clean
....:     def state(self):
....:         return "clean" if self.is_clean() else "dirty"
```

(continues on next page)
Now one can reuse this behavior within a class `Spoon`:

```python
def __repr__(self):
    return "An unspecified %s dish"%self.state()
def __repr__(self):
    return "A %s spoon"%self.state()
def eat_with(self):
    self._make_dirty()
def eat_with(self):
    self._make_dirty()
```

Let’s test it:

```python
s = Spoon(); s
A clean spoon
s.is_clean()
True
s.eat_with(); s
A dirty spoon
s.is_clean()
False
s.wash(); s
A clean spoon
```

**Summary**

1. Any class can reuse the behavior of another class. One says that the subclass **inherits** from the superclass or that it **derives** from it.

2. Any instance of the subclass is also an instance of its superclass:

```python
s = type(s)
<class '__main__.Spoon'>
```

```python
s.isinstance(s, Spoon)
True
```

```python
s.isinstance(s, AbstractDish)
True
```

3. If a subclass redefines a method, then it replaces the former one. One says that the subclass **overloads** the method. One can nevertheless explicitly call the hidden superclass method.

```python
s.__repr__()
'A clean spoon'
```

```python
Spoon.__repr__(s)
'A clean spoon'
```

```python
AbstractDish.__repr__(s)
'An unspecified clean dish'
```
Note: Advanced superclass method call

Sometimes one wants to call an overloaded method without knowing in which class it is defined. To do this, use the `super` operator:

```python
sage: super(Spoon, s).__repr__()
'An unspecified clean dish'
```

A very common usage of this construct is to call the `__init__` method of the superclass:

```python
sage: class Spoon(AbstractDish):
    ....:     def __init__(self):
    ....:         print("Building a spoon")
    ....:         super(Spoon, self).__init__()
....:
    ....:     def __repr__(self):
    ....:         return "A %s spoon"%self.state()
    ....:     def eat_with(self):
    ....:         self._make_dirty()

sage: s = Spoon()
Building a spoon
sage: s
A clean spoon
```

Exercises

1. Modify the class `Glasses` so that it inherits from `Dish`.
2. Write a class `Plate` whose instance can contain any meal together with a class `Fork`. Avoid as much as possible code duplication (hint: you can write a factorized class `ContainerDish`).
3. Test everything.

Sage specifics about classes

Compared to Python, Sage has particular ways to handle objects:

- Any classes for mathematical objects in Sage should inherit from `SageObject` rather than from `object`. Most of the time, they actually inherit from a subclass such as `Parent` or `Element`.
- Printing should be done through `_repr_` instead of `__repr__` to allow for renaming.
- More generally, Sage-specific special methods are usually named `_meth_` rather than `__meth__`. For example, lots of classes implement `_hash_` which is used and cached by `__hash__`. In the same vein, elements of a group usually implement `_mul_`, so that there is no need to take care about coercions as they are done in `__mul__`.

For more details, see the Sage Developer’s Guide.
Solutions to the exercises

1. Here is a solution to the first exercise:

```python
from __future__ import print_function

class Glass:
    def __init__(self, size):
        assert size > 0
        self._size = float(size)
        self.wash()
    
    def __repr__(self):
        if self._content == 0.0:
            return "An empty glass of size \%s\"%(self._size)
        else:
            return "A glass of size \%s cl containing \%s cl of \%s\"%(self._size, self._content, self._beverage)
    
    def content(self):
        return self._content
    
    def beverage(self):
        return self._beverage
    
    def fill(self, beverage = "water"):
        if not self.is_clean():
            raise ValueError("Don't want to fill a dirty glass")
        self._clean = False
        self._content = self._size
        self._beverage = beverage
    
    def empty(self):
        self._content = float(0.0)
    
    def is_empty(self):
        return self._content == 0.0
    
    def drink(self, amount):
        if amount <= 0.0:
            raise ValueError("amount must be positive")
        elif amount > self._content:
            raise ValueError("not enough beverage in the glass")
        else:
            self._content -= float(amount)
    
    def is_clean(self):
        return self._clean
    
    def wash(self):
        self._content = float(0.0)
        self._beverage = None
        self._clean = True
```

2. Let’s check that everything is working as expected:

```python
sage: G = Glass(10.0)
sage: G
An empty glass of size 10.0
sage: G.is_empty()  
True
sage: G.drink(2)   
Traceback (most recent call last):  
...
ValueError: not enough beverage in the glass
```
sage: G.fill("beer")
A glass of size 10.0 cl containing 10.0 cl of beer
sage: G.is_empty()
False
sage: G.is_clean()
False
sage: G.drink(5.0)
sage: G
A glass of size 10.0 cl containing 5.0 cl of beer
sage: G.is_empty()
False
sage: G.is_clean()
False
sage: G.drink(5)
sage: G
An empty glass of size 10.0
sage: G.is_clean()
False
sage: G.fill("orange juice")
Traceback (most recent call last):
...  
ValueError: Don't want to fill a dirty glass
sage: G.wash()
sage: G
An empty glass of size 10.0
sage: G.fill("orange juice")
sage: G
A glass of size 10.0 cl containing 10.0 cl of orange juice

3. Here is the solution to the second exercise:

```
sage: class AbstractDish():
    ....:     def __init__(self):
    ....:         self._clean = True
    ....:     def is_clean(self):
    ....:         return self._clean
    ....:     def state(self):
    ....:         return "clean" if self.is_clean() else "dirty"
    ....:     def __repr__(self):
    ....:         return "An unspecified %s dish"%self.state()
    ....:     def _make_dirty(self):
    ....:         self._clean = False
    ....:     def wash(self):
    ....:         self._clean = True

sage: class ContainerDish(AbstractDish):
    ....:     def __init__(self, size):
    ....:         assert size > 0
    ....:         self._size = float(size)
    ....:         self._content = float(0)
```

(continues on next page)
4. Let’s check that everything is working as expected:

```
sage: G = Glass(10.0)
sage: G
An empty glass of size 10.0
sage: G.is_empty()
True
sage: G.drink(2)
Traceback (most recent call last):
  ...: ValueError: not enough beverage in the glass
sage: G.fill("beer")
sage: G
A glass of size 10.0 cl containing 10.0 cl of beer
```

(continues on next page)
Todo:  give the example of the class Plate.

That all folks !

12.1.15 Functional Programming for Mathematicians

Author: Minh Van Nguyen <nguyenminh2@gmail.com>

This tutorial discusses some techniques of functional programming that might be of interest to mathematicians or people who use Python for scientific computation. We start off with a brief overview of procedural and object-oriented programming, and then discuss functional programming techniques. Along the way, we briefly review Python’s built-in support for functional programming, including filter, lambda, map and reduce. The tutorial concludes with some resources on detailed information on functional programming using Python.
Styles of programming

Python supports several styles of programming. You could program in the procedural style by writing a program as a list of instructions. Say you want to implement addition and multiplication over the integers. A procedural program to do so would be as follows:

```python
sage: def add_ZZ(a, b):
    ....:     return a + b
...

sage: def mult_ZZ(a, b):
    ....:     return a * b
...

sage: add_ZZ(2, 3)
5
sage: mult_ZZ(2, 3)
6
```

The Python module `operator` defines several common arithmetic and comparison operators as named functions. Addition is defined in the built-in function `operator.add` and multiplication is defined in `operator.mul`. The above example can be worked through as follows:

```python
sage: from operator import add
sage: from operator import mul

sage: add(2, 3)
5
sage: mul(2, 3)
6
```

Another common style of programming is called object-oriented programming. Think of an object as code that encapsulates both data and functionalities. You could encapsulate integer addition and multiplication as in the following object-oriented implementation:

```python
sage: class MyInteger:
    ....:     def __init__(self):
    ....:         self.cardinality = "infinite"
    ....:     def add(self, a, b):
    ....:         return a + b
    ....:     def mult(self, a, b):
    ....:         return a * b
...

sage: myZZ = MyInteger()
sage: myZZ.cardinality
'infinite'
sage: myZZ.add(2, 3)
5
sage: myZZ.mult(2, 3)
6
```
Functional programming using map

Functional programming is yet another style of programming in which a program is decomposed into various functions. The Python built-in functions map, reduce and filter allow you to program in the functional style. Note that in Python 3 (as compared to Python 2), these functions have different behaviors, and reduce has been removed: if you want to use reduce in Python 3, you must import it from functools.

The function

```
map(func, seq1, seq2, ...)
```

takes a function func and one or more sequences, and apply func to elements of those sequences. In particular, you end up with a list like so:

```
[func(seq1[0], seq2[0], ...), func(seq1[1], seq2[1], ...), ...]
```

In many cases, using map allows you to express the logic of your program in a concise manner without using list comprehension. For example, say you have two lists of integers and you want to add them element-wise. A list comprehension to accomplish this would be as follows:

```
sage: A = [1, 2, 3, 4]
sage: B = [2, 3, 5, 7]
sage: [A[i] + B[i] for i in range(len(A))]
[3, 5, 8, 11]
```

Alternatively, you could use the Python built-in addition function `operator.add` together with `map` to achieve the same result:

```
sage: from operator import add
sage: A = [1, 2, 3, 4]
sage: B = [2, 3, 5, 7]
sage: list(map(add, A, B))
[3, 5, 8, 11]
```

An advantage of `map` is that you do not need to explicitly define a for loop as was done in the above list comprehension.

Define small functions using lambda

There are times when you want to write a short, one-liner function. You could re-write the above addition function as follows:

```
sage: def add_ZZ(a, b): return a + b
...```

Or you could use a `lambda` statement to do the same thing in a much clearer style. The above addition and multiplication functions could be written using `lambda` as follows:

```
sage: add_ZZ = lambda a, b: a + b
sage: mult_ZZ = lambda a, b: a * b
sage: add_ZZ(2, 3)
5
sage: mult_ZZ(2, 3)
6
```
Things get more interesting once you combine `map` with the `lambda` statement. As an exercise, you might try to write a simple function that implements a constructive algorithm for the Chinese Remainder Theorem. You could use list comprehension together with `map` and `lambda` as shown below. Here, the parameter $A$ is a list of integers and $M$ is a list of moduli.

```python
sage: def crt(A, M):
    Mprod = prod(M)
    Mdiv = list(map(lambda x: Integer(Mprod / x), M))
    X = list(map(inverse_mod, Mdiv, M))
    x = sum([A[i]*X[i]*Mdiv[i] for i in range(len(A))])
    return mod(x, Mprod).lift()
```

```python
sage: A = [2, 3, 1]
sage: M = [3, 4, 5]
sage: x = crt(A, M); x
11
sage: mod(x, 3)
2
sage: mod(x, 4)
3
sage: mod(x, 5)
1
```

To produce a random matrix over a ring, say $\mathbb{Z}$, you could start by defining a matrix space and then obtain a random element of that matrix space:

```python
sage: MS = MatrixSpace(ZZ, nrows=5, ncols=3)
sage: MS.random_element() # random
```

```python
[ 6  1  0]
[-1  5  0]
[-1  0  0]
[-5  0  1]
[ 1 -1 -3]
```

Or you could use the function `random_matrix`:

```python
sage: random_matrix(ZZ, nrows=5, ncols=3) # random
```

```python
[ 2 -50  0]
[-1  0 -6]
[-4 -1 -1]
[ 1  1  3]
[ 2 -1 -1]
```

The next example uses `map` to construct a list of random integer matrices:

```python
sage: rows = [randint(1, 10) for i in range(10)]
sage: cols = [randint(1, 10) for i in range(10)]
sage: rings = [ZZ]*10
sage: M = list(map(random_matrix, rings, rows, cols))
```

```
[[-1 -3 -1 -37  1 -1 -4  5]]
```
If you want more control over the entries of your matrices than the `random_matrix` function permits, you could use `lambda` together with `map` as follows:

```python
sage: rand_row = lambda n: [randint(1, 10) for i in range(n)]

sage: rand_mat = lambda nrows, ncols: [rand_row(ncols) for i in range(nrows)]

sage: matrix(rand_mat(5, 3))  # random
[ 2  9 10]
[ 8  8  9]
[ 6  7  6]
[ 9  2 10]
[ 2  6  2]

sage: rows = [randint(1, 10) for i in range(10)]

sage: cols = [randint(1, 10) for i in range(10)]

sage: M = list(map(rand_mat, rows, cols))

sage: M = list(map(matrix, M))

sage: M[0]  # random
[ 9  1  5  2 10 10  1]
[ 3  4  3  7  4  3  7]
[ 4  8  7  6  4  2 10]
[ 1  6  3  6  2  1]
[ 5  5  2  6  4  3  4]
[ 6  6  2  9  4  5  1]
[10  2  5  5  7 10  4]
[ 2  7  3  5 10  8  1]
[ 1  5  1  7  8  8  6]
```

### Reducing a sequence to a value

The function `reduce` takes a function of two arguments and apply it to a given sequence to reduce that sequence to a single value. The function `sum` is an example of a `reduce` function. The following sample code uses `reduce` and the built-in function `operator.add` to add together all integers in a given list. This is followed by using `sum` to accomplish the same task:

```python
sage: from functools import reduce

sage: L = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

sage: reduce(add, L)
55

sage: sum(L)
55
```

In the following sample code, we consider a vector as a list of real numbers. The dot product is then implemented using the functions `operator.add` and `operator.mul`, in conjunction with the built-in Python functions `reduce` and `map`. We then show how `sum` and `map` could be combined to produce the same result.
Or you could use Sage’s built-in support for the dot product:

```
sage: u = vector([1, 2, 3])
sage: v = vector([2, 3, 5])
sage: u.dot_product(v)
23
```

Here is an implementation of the Chinese Remainder Theorem without using `sum` as was done previously. The version below uses `operator.add` and defines `mul3` to multiply three numbers instead of two.

```
sage: from functools import reduce
sage: def crt(A, M):
         ....:     from operator import add
         ....:     Mprod = prod(M)
         ....:     Mdiv = list(map(lambda x: Integer(Mprod / x), M))
         ....:     X = map(inverse_mod, Mdiv, M)
         ....:     mul3 = lambda a, b, c: a * b * c
         ....:     x = reduce(add, map(mul3, A, X, Mdiv))
         ....:     return mod(x, Mprod).lift()
...

sage: A = [2, 3, 1]
sage: M = [3, 4, 5]
sage: x = crt(A, M); x
11
```

Filtering with filter

The Python built-in function `filter` takes a function of one argument and a sequence. It then returns a list of all those items from the given sequence such that any item in the new list results in the given function returning `True`. In a sense, you are filtering out all items that satisfy some condition(s) defined in the given function. For example, you could use `filter` to filter out all primes between 1 and 50, inclusive.

```
sage: list(filter(is_prime, [1..50]))
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

For a given positive integer \( n \), the Euler phi function counts the number of integers \( a \), with 1 ≤ \( a \) ≤ \( n \), such that \( \gcd(a, n) = 1 \). You could use list comprehension to obtain all such \( a \)'s when \( n = 20 \):

```
sage: [k for k in range(1, 21) if gcd(k, 20) == 1]
[1, 3, 7, 9, 11, 13, 17, 19]
```

A functional approach is to use `lambda` to define a function that determines whether or not a given integer is relatively prime to 20. Then you could use `filter` instead of list comprehension to obtain all the required \( a \)'s.
The function `primroots` defined below returns all primitive roots modulo a given positive prime integer $p$. It uses `filter` to obtain a list of integers between 1 and $p - 1$, inclusive, each integer in the list being relatively prime to the order of the multiplicative group $(\mathbb{Z}/p\mathbb{Z})^*$. 

```python
sage: def primroots(p):
....:     g = primitive_root(p)
....:     znorder = p - 1
....:     is_coprime = lambda x: gcd(x, znorder) == 1
....:     good_odd_integers = filter(is_coprime, [1..p-1, step=2])
....:     all_primroots = [power_mod(g, k, p) for k in good_odd_integers]
....:     all_primroots.sort()
....:     return all_primroots
...
```

```
[2]
sage: primroots(5) [2, 3]
sage: primroots(7) [3, 5]
sage: primroots(11) [2, 6, 7, 8]
sage: primroots(13) [2, 6, 7, 11]
sage: primroots(17) [3, 5, 6, 7, 10, 11, 12, 14]
sage: primroots(23) [5, 7, 10, 11, 14, 15, 17, 19, 20, 21]
sage: primroots(29) [2, 3, 8, 10, 11, 14, 15, 18, 19, 21, 26, 27]
sage: primroots(31) [3, 11, 12, 13, 17, 21, 22, 24]
```

Further resources

This has been a rather short tutorial to functional programming with Python. The Python standard documentation has a list of built-in functions, many of which are useful in functional programming. For example, you might want to read up on `all`, `any`, `max`, `min`, and `zip`. The Python module `operator` has numerous built-in arithmetic and comparison operators, each operator being implemented as a function whose name reflects its intended purpose. For arithmetic and comparison operations, it is recommended that you consult the `operator` module to determine if there is a built-in function that satisfies your requirement. The module `itertools` has numerous built-in functions to efficiently process sequences of items.

Another useful resource for functional programming in Python is the Functional Programming HOWTO by A. M. Kuchling. Steven F. Lott’s book Building Skills in Python has a chapter on Functional Programming using Collections. See also the chapter Functional Programming from Mark Pilgrim’s book Dive Into Python.

You might also want to consider experimenting with Haskell for expressing mathematical concepts. For an example of Haskell in expressing mathematical algorithms, see J. Gibbons’ article Unbounded Spigot Algorithms for the Digits of Pi in the American Mathematical Monthly.
12.1.16 How to implement new algebraic structures in Sage

Contents

- How to implement new algebraic structures in Sage
  - Sage’s category and coercion framework
    * Outline
    * Base classes
    * Categories in Sage
    * Coercion—the basics
    * Coercion—the advanced parts
    * The test suites of the category framework
    * Appendix: The complete code

Sage’s category and coercion framework

Author: Simon King, Friedrich-Schiller-Universität Jena, <simon.king@uni-jena.de> © 2011/2013

The aim of this tutorial is to explain how one can benefit from Sage’s category framework and coercion model when implementing new algebraic structures. It is based on a worksheet created in 2011.

We illustrate the concepts of Sage’s category framework and coercion model by means of a detailed example, namely a toy implementation of fraction fields. The code is developed step by step, so that the reader can focus on one detail in each part of this tutorial. The complete code can be found in the appendix.

Outline

- Use existing base classes
  
  For using Sage’s coercion system, it is essential to work with sub-classes of sage.structure.parent.Parent or sage.structure.element.Element, respectively. They provide default implementations of many “magical” double-underscore Python methods, which must not be overridden. Instead, the actual implementation should be in single underscore methods, such as _add_ or _mul_.
- Turn your parent structure into an object of a category
  
  Declare the category during initialisation—Your parent structure will inherit further useful methods and consistency tests.
- Provide your parent structure with an element class
  
  Assign to it an attribute called Element—The elements will inherit further useful methods from the category. In addition, some basic conversions will immediately work.
- Implement further conversions
  
  Never override a parent’s __call__ method! Provide the method _element_constructor_ instead.
- Declare coercions
  
  If a conversion happens to be a morphism, you may consider to turn it into a coercion. It will then implicitly be used in arithmetic operations.
• Advanced coercion: Define construction functors for your parent structure

Sage will automatically create new parents for you when needed, by the so-called sage.categories.pushout.pushout() construction.

• Run the automatic test suites

Each method should be documented and provide a doc test (we are not giving examples here). In addition, any method defined for the objects or elements of a category should be supported by a test method, that is executed when running the test suite.

Base classes

In Sage, a “Parent” is an object of a category and contains elements. Parents should inherit from sage.structure.parent.Parent and their elements from sage.structure.element.Element.

Sage provides appropriate sub-classes of Parent and Element for a variety of more concrete algebraic structures, such as groups, rings, or fields, and of their elements. But some old stuff in Sage doesn’t use it. **Volunteers for refactoring are welcome!**

The parent

Since we wish to implement a special kind of fields, namely fraction fields, it makes sense to build on top of the base class sage.rings.ring.Field provided by Sage.

```sage
from sage.rings.ring import Field
```

This base class provides a lot more methods than a general parent:

```sage
[p for p in dir(Field) if p not in dir(Parent)]
['__fraction_field',
 '__ideal_monoid',
 '__iter__',
 '__len__',
 '__rtruediv__',
 '__rxor__',
 '__truediv__',
 '__xor__',
 '_an_element_impl',
 '_coerce_',
 '_coerce_c',
 '_coerce_impl',
 '_coerce_try',
 '_default_category',
 '_gens',
 '_ideal_class_',
 '_latex_names',
 '_list',
 '_one_element',
 '_pseudo_fraction_field',
 '_random_nonzero_element',
 '_unit_ideal',
 '_zero_element',
 '_zero_ideal',
```

(continues on next page)
Thematic Tutorials, Release 9.7

The following is a very basic implementation of fraction fields, that needs to be complemented later.

```python
sage: from sage.structure.unique_representation import UniqueRepresentation
sage: class MyFrac(UniqueRepresentation, Field):
    ....:     def __init__(self, base):
    ....:         if base not in IntegralDomains():
    ....:             raise ValueError("%s is no integral domain" % base)
    ....:         Field.__init__(self, base)
    ....:     def _repr_(self):
    ....:         return "NewFrac(%s)"%repr(self.base())
    ....:     def base_ring(self):
```

(continues on next page)
This basic implementation is formed by the following steps:

- Any ring in Sage has a **base** and a **base ring**. The “usual” fraction field of a ring $R$ has the base $R$ and the base ring $R\text{.base()}$:

```
sage: Frac(QQ['x']).base(), Frac(QQ['x']).base_ring()
(Univariate Polynomial Ring in x over Rational Field, Rational Field)
```

Declaring the base is easy: We just pass it as an argument to the field constructor.

```
sage: Field(ZZ['x']).base()
Univariate Polynomial Ring in x over Integer Ring
```

We are implementing a separate method returning the base ring.

- Python uses double--underscore methods for arithmetic methods and string representations. Sage’s base classes often have a default implementation, and it is requested to implement **SINGLE underscore methods** _repr_, and similarly _add_, _mul_ etc.

- You are encouraged to make your parent “unique”. That’s to say, parents should only evaluate equal if they are identical. Sage provides frameworks to create unique parents. We use here the most easy one: Inheriting from the class `sage.structure.unique_representation.UniqueRepresentation` is enough. Making parents unique can be quite important for an efficient implementation, because the repeated creation of “the same” parent would take a lot of time.

- Fraction fields are only defined for integral domains. Hence, we raise an error if the given ring does not belong to the category of integral domains. This is our first use case of categories.

- Last, we add a method that returns the characteristic of the field. We don’t go into details, but some automated tests that we study below implicitly rely on this method.

We see that our basic implementation correctly refuses a ring that is not an integral domain:

```
sage: MyFrac(ZZ['x'])
NewFrac(Univariate Polynomial Ring in x over Integer Ring)
sage: MyFrac(Integers(15))
Traceback (most recent call last):
... ValueError: Ring of integers modulo 15 is no integral domain
```

**Note:** Inheritance from `UniqueRepresentation` automatically provides our class with pickling, preserving the unique parent condition. If we had defined the class in some external module or in an interactive session, pickling would work immediately.

However, for making the following example work in Sage’s doctesting framework, we need to assign our class as an attribute of the `__main__` module, so that the class can be looked up during unpickling.
Note: In the following sections, we will successively add or change details of MyFrac. Rather than giving a full class definition in each step, we define new versions of MyFrac by inheriting from the previously defined version of MyFrac. We believe this will help the reader to focus on the single detail that is relevant in each section.

The complete code can be found in the appendix.

The elements

We use the base class `sage.structure.element.FieldElement`. Note that in the creation of field elements it is not tested that the given parent is a field:

```
sage: from sage.structure.element import FieldElement
sage: FieldElement(ZZ)
Generic element of a structure
```

Our toy implementation of fraction field elements is based on the following considerations:

- A fraction field element is defined by numerator and denominator, which both need to be elements of the base. There should be methods returning numerator resp. denominator.

- The denominator must not be zero, and (provided that the base is an ordered ring) we can make it non-negative, without loss of generality. By default, the denominator is one.

- The string representation is returned by the single-underscore method `_repr_`. In order to make our fraction field elements distinguishable from those already present in Sage, we use a different string representation.

- Arithmetic is implemented in single-underscore method `_add_`, `_mul_`, etc. We do not override the default double underscore `__add__`, `__mul__`, since otherwise, we could not use Sage’s coercion model.

- Comparisons can be implemented using `_richcmp_`. This automatically makes the relational operators like `==` and `<` work. Inside this method, you can use the `richcmp` functions and related tools provided by sage.

Note that `_richcmp_` should be provided, since otherwise comparison does not work:

```
sage: class Foo(sage.structure.element.Element):
    ....:     def __init__(self, parent, x):
    ....:         self.x = x
    ....:     def _repr_(self):
    ....:         return '<%s>' % self.x
sage: a = Foo(ZZ, 1)
sage: b = Foo(ZZ, 2)
sage: a <= b
Traceback (most recent call last):
  ... TypeError: '<=' not supported between instances of 'Foo' and 'Foo'
```

- In the single underscore methods, we can assume that both arguments belong to the same parent. This is one benefit of the coercion model.

- When constructing new elements as the result of arithmetic operations, we do not directly name our class, but we use `self.__class__`. Later, this will come in handy.

This gives rise to the following code:
```python
sage: class MyElement(FieldElement):
    ....:     def __init__(self, parent, n, d=None):
    ....:         B = parent.base()
    ....:         if d is None:
    ....:             d = B.one()
    ....:         if n not in B or d not in B:
    ....:             raise ValueError("Numerator and denominator must be elements of %s"%B)
    ....:         # Numerator and denominator should not just be "in" B,
    ....:         # but should be defined as elements of B
    ....:         d = B(d)
    ....:         n = B(n)
    ....:         if d==0:
    ....:             raise ZeroDivisionError("The denominator must not be zero")
    ....:         if d<0:
    ....:             self.n = -n
    ....:             self.d = -d
    ....:         else:
    ....:             self.n = n
    ....:             self.d = d
    FieldElement.__init__(self,parent)
    ....:     def numerator(self):
    ....:         return self.n
    ....:     def denominator(self):
    ....:         return self.d
    ....:     def __repr__(self):
    ....:         return "(%s):(%s)"%(self.n, self.d)
    ....:     def __richcmp__(self, other, op):
    ....:         from sage.structure.richcmp import richcmp
    ....:         return richcmp(self.n*other.denominator(), other.numerator()*self.d, op)
    ....:     def __add__(self, other):
    ....:         C = self.__class__
    ....:         D = self.d*other.denominator()
    ....:         return C(self.parent(), self.n*other.denominator()+self.d*other.numerator(), D)
    ....:     def __sub__(self, other):
    ....:         C = self.__class__
    ....:         D = self.d*other.denominator()
    ....:         return C(self.parent(), self.n*other.denominator()-self.d*other.numerator(), D)
    ....:     def __mul__(self, other):
    ....:         C = self.__class__
    ....:         return C(self.parent(), self.n*other.numerator(), self.d*other.denominator())
    ....:     def __div__(self, other):
    ....:         C = self.__class__
    ....:         return C(self.parent(), self.n*other.denominator(), self.d*other.numerator())
```

12.1. Thematic tutorial document tree
Features and limitations of the basic implementation

Thanks to the single underscore methods, some basic arithmetics works, if we stay inside a single parent structure:

```sage
sage: P = MyFrac(ZZ)
sage: a = MyElement(P, 3, 4)
sage: b = MyElement(P, 1, 2)
sage: a+b, a-b, a*b, a/b
((10):(8), (2):(8), (3):(8), (6):(4))
sage: a-b == MyElement(P, 1, 4)
True
```

There is a default implementation of element tests. We can already do

```sage
sage: a in P
True
```

since `a` is defined as an element of `P`. However, we cannot verify yet that the integers are contained in the fraction field of the ring of integers. It does not even give a wrong answer, but results in an error:

```sage
sage: 1 in P
Traceback (most recent call last):
...    NotImplementedError: cannot construct elements of NewFrac(Integer Ring)
```

We will take care of this later.

Categories in Sage

Sometimes the base classes do not reflect the mathematics: The set of \( m \times n \) matrices over a field forms, in general, not more than a vector space. Hence, this set (called `MatrixSpace`) is not implemented on top of `sage.rings.ring.Ring`. However, if \( m = n \), then the matrix space is an algebra, thus, is a ring.

From the point of view of Python base classes, both cases are the same:

```sage
sage: MS1 = MatrixSpace(QQ,2,3)
sage: isinstance(MS1, Ring)
False
sage: MS2 = MatrixSpace(QQ,2)
sage: isinstance(MS2, Ring)
False
```

Sage’s category framework can differentiate the two cases:

```sage
sage: Rings()
Category of rings
sage: MS1 in Rings()
False
sage: MS2 in Rings()
True
```

And indeed, `MS2` has more methods than `MS1`:
This is because the class of $\texttt{MS2}$ also inherits from the parent class for algebras:

\begin{verbatim}
sage: MS1.__class__.__bases__
(<class 'sage.matrix.matrix_space.MatrixSpace'>,
 <class 'sage.categories.category.JoinCategory.parent_class'>)

sage: MS2.__class__.__bases__
(<class 'sage.matrix.matrix_space.MatrixSpace'>,
 <class 'sage.categories.category.JoinCategory.parent_class'>)
\end{verbatim}

Below, we will explain how this can be taken advantage of.

It is no surprise that our parent $P$ defined above knows that it belongs to the category of fields, as it is derived from the base class of fields.

\begin{verbatim}
sage: P.category()
Category of fields
\end{verbatim}

However, we could choose a smaller category, namely the category of quotient fields.

Why should one choose a category?

One can provide default methods for all objects of a category, and for all elements of such objects. Hence, the category framework is a way to inherit useful stuff that is not present in the base classes. These default methods do not rely on implementation details, but on mathematical concepts.

In addition, the categories define test suites for their objects and elements—see the last section. Hence, one also gets basic sanity tests for free.

How does the category framework work?

Abstract base classes for the objects (“parent_class”) and the elements of objects (“element_class”) are provided by attributes of the category. During initialisation of a parent, the class of the parent is dynamically changed into a sub-class of the category’s parent class. Likewise, sub-classes of the category’s element class are available for the creation of elements of the parent, as explained below.

A dynamic change of classes does not work in Cython. Nevertheless, method inheritance still works, by virtue of a __getattr__ method.

Note: It is strongly recommended to use the category framework both in Python and in Cython.

Let us see whether there is any gain in choosing the category of quotient fields instead of the category of fields:

\begin{verbatim}
[continues on next page]
\end{verbatim}
So, there is no immediate gain for our fraction fields, but additional methods become available to our fraction field elements. Note that some of these methods are place-holders: There is no default implementation, but it is \textit{required} (respectively is \textit{optional}) to implement these methods:

\begin{verbatim}
[sage: QuotientFields().element_class.denominator
<abstract method denominator at ...>
]
sage: from sage.misc.abstract_method import abstract_methods_of_class
sage: abstract_methods_of_class(QuotientFields().element_class)["optional"]
[['__add__', '__mul__']]
sage: abstract_methods_of_class(QuotientFields().element_class)["required"]
[['__bool__', 'denominator', 'numerator']]
\end{verbatim}

Hence, when implementing elements of a quotient field, it is \textit{required} to implement methods returning the denominator and the numerator, and a method that tells whether the element is nonzero, and in addition, it is \textit{optional} (but certainly recommended) to provide some arithmetic methods. If one forgets to implement the required methods, the test suites of the category framework will complain—see below.

\section*{Implementing the category framework for the parent}

We simply need to declare the correct category by an optional argument of the field constructor, where we provide the possibility to override the default category:

\begin{verbatim}
sage: from sage.categories.quotient_fields import QuotientFields
sage: class MyFrac(MyFrac):
....:     def __init__(self, base, category=None):
....:         if base not in IntegralDomains():
....:             raise ValueError("%s is no integral domain" % base)
....:         Field.__init__(self, base, category=category or QuotientFields())
\end{verbatim}

When constructing instances of \texttt{MyFrac}, their class is dynamically changed into a new class called \texttt{MyFrac_with_category}. It is a common sub-class of \texttt{MyFrac} and of the category's parent class:

\begin{verbatim}
sage: P = MyFrac(ZZ)
sage: type(P)
<class '__main__.MyFrac_with_category'>
sage: isinstance(P,MyFrac)
True
sage: isinstance(P,QuotientFields().parent_class)
True
\end{verbatim}

The fraction field \(P\) inherits additional methods. For example, the base class \texttt{Field} does not have a method \texttt{sum}. But \(P\) inherits such method from the category of commutative additive monoids—see \texttt{sum()}:
We have seen above that we can add elements. Nevertheless, the sum method does not work, yet:

```
sage: a = MyElement(P, 3, 4)
sage: b = MyElement(P, 1, 2)
sage: c = MyElement(P, -1, 2)
sage: P.sum([a, b, c])
Traceback (most recent call last):
...
NotImplementedError: cannot construct elements of NewFrac(Integer Ring)
```

The reason is that the sum method starts with the return value of \( P \text{.zero}() \), which defaults to \( P(0) \)---but the conversion of integers into \( P \) is not implemented, yet.

### Implementing the category framework for the elements

Similar to what we have seen for parents, a new class is dynamically created that combines the element class of the parent’s category with the class that we have implemented above. However, the category framework is implemented in a different way for elements than for parents:

- We provide the parent \( P \) (or its class) with an attribute called “Element”, whose value is a class.
- The parent automatically obtains an attribute \( P \text{.element_class} \), that subclasses both \( P \text{.Element} \) and \( P \text{.category()\text{.element_class}} \).

Hence, for providing our fraction fields with their own element classes, we just need to add a single line to our class:

```
sage: class MyFrac(MyFrac):
....:   Element = MyElement
```

This little change provides several benefits:

- We can now create elements by simply calling the parent:

```
sage: P = MyFrac(ZZ)
sage: P(1), P(2,3)
((1):(1), (2):(3))
```

- There is a method zero returning the expected result:

```
sage: P.zero()
(0):(1)
```

- The sum method mentioned above suddenly works:

```
sage: a = MyElement(P, 9, 4)
sage: b = MyElement(P, 1, 2)
sage: c = MyElement(P, -1, 2)
sage: P.sum([a,b,c])
(36):(16)
```

- Exponentiation now works out of the box using the multiplication that we defined:
What did happen behind the scenes to make this work?

We provided `P.Element`, and thus obtain `P.element_class`, which is a `lazy attribute`. It provides a `dynamic` class, which is a sub-class of both `MyElement` defined above and of `P.category().element_class`:

```
sage: P.__class__.element_class
class '__main__.MyFrac_with_category.element_class'
sage: type(P.element_class)
class 'sage.structure.dynamic_class.DynamicInheritComparisonMetaclass'
sage: issubclass(P.element_class, MyElement)
True
sage: issubclass(P.element_class,P.category().element_class)
True
```

The `default __call__` method of `P` passes the given arguments to `P.element_class`, adding the argument `parent=P`. This is why we are now able to create elements by calling the parent.

In particular, these elements are instances of that new dynamic class:

```
sage: type(P(2,3))
class '__main__.MyFrac_with_category.element_class'
```

**Note:** All elements of `P` should use the element class. In order to make sure that this also holds for the result of arithmetic operations, we created them as instances of `self.__class__` in the arithmetic methods of `MyElement`.

`P.zero()` defaults to returning `P(0)` and thus returns an instance of `P.element_class`. Since `P.sum([...])` starts the summation with `P.zero()` and the class of the sum only depends on the first summand, by our implementation, we have:

```
sage: type(a)
class '__main__.MyElement'
sage: isinstance(a,P.element_class)
False
sage: type(P.sum([a,b,c]))
class '__main__.MyFrac_with_category.element_class'
```

The method `factor` provided by `P.category().element_class` (see above) simply works:

```
sage: a; a.factor(); P(6,4).factor()
(9):(4)
2^-2 * 3^2
2^-1 * 3
```

But that’s surprising: The element `a` is just an instance of `MyElement`, but not of `P.element_class`, and its class does not know about the factor method. In fact, this is due to a `__getattr__` method defined for `sage.structure.element.Element`. 

410 Chapter 12. Documentation
### A first note on performance

The category framework is sometimes blamed for speed regressions, as in trac ticket #9138 and trac ticket #11900. But if the category framework is used properly, then it is fast. For illustration, we determine the time needed to access an attribute inherited from the element class. First, we consider an element that uses the class that we implemented above, but does not use the category framework properly:

```python
sage: type(a)
<class '__main__.MyElement'>
sage: timeit('a.factor',number=1000)    # random
1000 loops, best of 3: 2 us per loop
```

Now, we consider an element that is equal to \(a\), but uses the category framework properly:

```python
sage: a2 = P(9,4)
sage: a2 == a
True
sage: type(a2)
<class '__main__.MyFrac_with_category.element_class'>
sage: timeit('a2.factor',number=1000)    # random
1000 loops, best of 3: 365 ns per loop
```

So, don’t be afraid of using categories!

### Coercion—the basics

#### Theoretical background

**Coercion is not just type conversion**

“Coercion” in the C programming language means “automatic type conversion”. However, in Sage, coercion is involved if one wants to be able to do arithmetic, comparisons, etc. between elements of distinct parents. Hence, coercion is not about a change of types, but about a change of parents.

As an illustration, we show that elements of the same type may very well belong to rather different parents:

```python
sage: P1 = QQ['v,w']; P2 = ZZ['w,v']
sage: type(P1.gen()) == type(P2.gen())
True
sage: P1 == P2
False
```

\(P_2\) naturally is a sub--ring of \(P_1\). So, it makes sense to be able to add elements of the two rings—the result should then live in \(P_1\), and indeed it does:
It would be rather inconvenient if one needed to manually convert an element of $P_2$ into $P_1$ before adding. The coercion system does that conversion automatically.

**Not every conversion is a coercion**

A coercion happens implicitly, without being explicitly requested by the user. Hence, coercion must be based on mathematical rigour. In our example, any element of $P_2$ can be naturally interpreted as an element of $P_1$. We thus have:

```python
sage: P1.has_coerce_map_from(P2)
True
sage: P1.coerce_map_from(P2)
Coercion map:
    From: Multivariate Polynomial Ring in w, v over Integer Ring
    To:    Multivariate Polynomial Ring in v, w over Rational Field
```

While there is a conversion from $P_1$ to $P_2$ (namely restricted to polynomials with integral coefficients), this conversion is not a coercion:

```python
sage: P2.convert_map_from(P1)
Conversion map:
    From: Multivariate Polynomial Ring in v, w over Rational Field
    To:    Multivariate Polynomial Ring in w, v over Integer Ring
sage: P2.has_coerce_map_from(P1)
False
sage: P2.coerce_map_from(P1) is None
True
```

**The four axioms requested for coercions**

1. A coercion is a morphism in an appropriate category.
   
   This first axiom has two implications:
   
   A. A coercion is defined on all elements of a parent.
      
      A polynomial of degree zero over the integers can be interpreted as an integer— but the attempt to convert a polynomial of non-zero degree would result in an error:
      
      ```python
      sage: ZZ(P2.one())
      1
      sage: ZZ(P2.gen(1))
      Traceback (most recent call last):
      ...
      TypeError: not a constant polynomial
      ```
      
      Hence, we only have a partial map. This is fine for a conversion, but a partial map does not qualify as a coercion.
      
   B. Coercions are structure preserving.
Any real number can be converted to an integer, namely by rounding. However, such a conversion is not useful in arithmetic operations, since the underlying algebraic structure is not preserved:

```
sage: int(1.6)+int(2.7) == int(1.6+2.7)
False
```

The structure that is to be preserved depends on the category of the involved parents. For example, the coercion from the integers into the rational field is a homomorphism of euclidean domains:

```
sage: QQ.coerce_map_from(ZZ).category_for()
Join of Category of euclidean domains and Category of infinite sets and Category of metric spaces
```

2. There is at most one coercion from one parent to another

In addition, if there is a coercion from $P_2$ to $P_1$, then a conversion from $P_2$ to $P_1$ is defined for all elements of $P_2$ and coincides with the coercion. Nonetheless, user-exposed maps are copies of the internally used maps whence the lack of identity between different instantiations:

```
sage: P1.coerce_map_from(P2) is P1.convert_map_from(P2)
False
```

For internally used maps, the maps are identical:

```
sage: P1._internal_coerce_map_from(P2) is P1._internal_convert_map_from(P2)
True
```

3. Coercions can be composed

If there is a coercion $\varphi : P_1 \rightarrow P_2$ and another coercion $\psi : P_2 \rightarrow P_3$, then the composition of $\varphi$ followed by $\psi$ must yield the unique coercion from $P_1$ to $P_3$.

4. The identity is a coercion

Together with the two preceding axioms, it follows: If there are coercions from $P_1$ to $P_2$ and from $P_2$ to $P_1$, then they are mutually inverse.

**Implementing a conversion**

We have seen above that some conversions into our fraction fields became available after providing the attribute `Element`. However, we cannot convert elements of a fraction field into elements of another fraction field, yet:

```
sage: P(2/3)
Traceback (most recent call last):
...
ValueError: Numerator and denominator must be elements of Integer Ring
```

For implementing a conversion, the default `__call__` method should (almost) never be overridden. Instead, we implement the method `__element_constructor__`, that should return an instance of the parent’s element class. Some old parent classes violate that rule—please help to refactor them!

```
sage: class MyFrac(MyFrac):
    ....:     def __element_constructor__(self, *args, **kwds):
    ....:         if len(args)!=1:
    ....:             return self.element_class(self, *args, **kwds)
```

(continues on next page)
In addition to the conversion from the base ring and from pairs of base ring elements, we now also have a conversion from the rationals to our fraction field of \( \mathbb{Z} \):

```python
sage: P = MyFrac(ZZ)
sage: P(2); P(2,3); P(3/4)
(2):(1)
(2):(3)
(3):(4)
```

Recall that above, the test \( 1 \in P \) failed with an error. We try again and find that the error has disappeared. This is because we are now able to convert the integer 1 into \( P \). But the containment test still yields a wrong answer:

```python
sage: 1 in P
False
```

The technical reason: We have a conversion \( P(1) \) of 1 into \( P \), but this is not known as a coercion—yet!

```python
sage: P.has_coerce_map_from(ZZ), P.has_coerce_map_from(QQ)
(False, False)
```

### Establishing a coercion

There are two main ways to make Sage use a particular conversion as a coercion:

- One can use `sage.structure.parent.Parent.register_coercion()`, normally during initialisation of the parent (see documentation of the method).
- A more flexible way is to provide a method `_coerce_map_from_` for the parent.

Let \( P \) and \( R \) be parents. If \( P._coerce_map_from_(R) \) returns `False` or `None`, then there is no coercion from \( R \) to \( P \). If it returns a map with domain \( R \) and codomain \( P \), then this map is used for coercion. If it returns `True`, then the conversion from \( R \) to \( P \) is used as coercion.

Note that in the following implementation, we need a special case for the rational field, since \( QQ.base() \) is not the ring of integers.

```python
sage: class MyFrac(MyFrac):
    ....:    def _coerce_map_from_(self, S):
    ....:        if self.base().has_coerce_map_from(S):
    ....:            return True
    ....:        if S in QuotientFields():
    ....:            if self.base().has_coerce_map_from(S.base()):
    ....:                return True
```
By the method above, a parent coercing into the base ring will also coerce into the fraction field, and a fraction field coerces into another fraction field if there is a coercion of the corresponding base rings. Now, we have:

```
sage: P = MyFrac(QQ['x'])
sage: P.has_coerce_map_from(ZZ['x']), P.has_coerce_map_from(Frac(ZZ['x'])), P.has_coerce_map_from(QQ)
(True, True, True)
```

We can now use coercion from $\mathbb{Z}[x]$ and from $\mathbb{Q}$ into $P$ for arithmetic operations between the two rings:

```
sage: 3/4+P(2)+ZZ['x'].gen(), (P(2)+ZZ['x'].gen()).parent() is P
((4*x + 11):(4), True)
```

### Equality and element containment

Recall that above, the test $1 \in P$ gave a wrong answer. Let us repeat the test now:

```
sage: 1 in P
True
```

Why is that?

The default element containment test $x \in P$ is based on the interplay of three building blocks: conversion, coercion, and equality test.

1. Clearly, if the conversion $P(x)$ raises an error, then $x$ cannot be seen as an element of $P$. On the other hand, a conversion $P(x)$ can generally do very nasty things. So, the fact that $P(x)$ works without error is necessary, but not sufficient for $x \in P$.
2. If $P$ is the parent of $x$, then the conversion $P(x)$ will not change $x$ (at least, that’s the default). Hence, we will have $x = P(x)$.
3. Sage uses coercion not only for arithmetic operations, but also for comparison: If there is a coercion from the parent of $x$ to $P$, then the equality test $x==P(x)$ reduces to $P(x)==P(x)$. Otherwise, $x==P(x)$ will evaluate as false.

That leads to the following default implementation of element containment testing:

---

**Note:** $x \in P$ holds if and only if the test $x==P(x)$ does not raise an error and evaluates as true.

If the user is not happy with that behaviour, the “magical” Python method `__contains__` can be overridden.
Coercion—the advanced parts

So far, we are able to add integers and rational numbers to elements of our new implementation of the fraction field of \( \mathbb{Z} \).

\[
\text{sage: } P = \text{MyFrac}(\mathbb{Z})
\]

\[
\text{sage: } 1/2 + P(2,3)+1
\]

\((13):(6)\)

Surprisingly, we can even add a polynomial over the integers to an element of \( P \), even though the result lives in a new parent, namely in a polynomial ring over \( \mathbb{Z} \):

\[
\text{sage: } P(1/2) + \mathbb{Z}[x].\text{gen()}, (P(1/2) + \mathbb{Z}[x].\text{gen()}).\text{parent()} \text{ is } P[x]
\]

\((1):(1)*x + (1):(2), \text{True}\)

In the next, seemingly more easy example, there “obviously” is a coercion from the fraction field of \( \mathbb{Z} \) to the fraction field of \( \mathbb{Z}[x] \). However, Sage does not know enough about our new implementation of fraction fields. Hence, it does not recognise the coercion:

\[
\text{sage: } \text{Frac}(\mathbb{Z}[x]).\text{has_coerce_map_from}(P)
\]

\(\text{False}\)

Two obvious questions arise:

1. How / why has the new ring been constructed in the example above?
2. How can we establish a coercion from \( P \) to \( \text{Frac}(\mathbb{Z}[x]) \)?

The key to answering both question is the construction of parents from simpler pieces, that we are studying now. Note that we will answer the second question not by providing a coercion from \( P \) to \( \text{Frac}(\mathbb{Z}[x]) \), but by teaching Sage to automatically construct \( \text{MyFrac}(\mathbb{Z}[x]) \) and coerce both \( P \) and \( \text{Frac}(\mathbb{Z}[x]) \) into it.

If we are lucky, a parent can tell how it has been constructed:

\[
\text{sage: } \text{Poly},R = \mathbb{Q}[x].\text{construction()}
\]

\(\text{sage: } \text{Poly},R
\)

(\(\text{Poly}[x] \), \text{Rational Field})

\[
\text{sage: } \text{Fract},R = \mathbb{Q}.\text{construction()}
\]

\(\text{sage: } \text{Fract},R
\)

(\text{FractionField}, \text{Integer Ring})

In both cases, the first value returned by \text{construction()} is a mathematical construction, called \textit{construction functor}—see \texttt{ConstructionFunctor}. The second return value is a simpler parent to which the construction functor is applied. Being functors, the same construction can be applied to different objects of a category:

\[
\text{sage: } \text{Poly}() \text{ is } \mathbb{Q}[x]
\]

\(\text{True}\)

\[
\text{sage: } \text{Poly}() \text{ is } \mathbb{Z}[x]
\]

\(\text{True}\)

\[
\text{sage: } \text{Poly}() \text{ is } P[x]
\]

\(\text{True}\)

\[
\text{sage: } \text{Frac}() \text{ is } \text{Fraction Field of Univariate Polynomial Ring in x over Rational Field}
\]

Let us see on which categories these construction functors are defined:
In particular, the construction functors can be composed:

```
sage: Poly*Fract
Poly[x](FractionField(...))
sage: (Poly*Fract)(ZZ) is QQ['x']
True
```

In addition, it is often assumed that we have a coercion from input to output of the construction functor:

```
sage: ((Poly*Fract)(ZZ)).coerce_map_from(ZZ)
Composite map:
  From: Integer Ring
  To:   Univariate Polynomial Ring in x over Rational Field
  Defn: Natural morphism:
        From: Integer Ring
        To:   Rational Field
        then
        Polynomial base injection morphism:
        From: Rational Field
        To:   Univariate Polynomial Ring in x over Rational Field
```

Construction functors do not necessarily commute:

```
sage: (Fract*Poly)(ZZ)
Fraction Field of Univariate Polynomial Ring in x over Integer Ring
```

### The pushout of construction functors

We can now formulate our problem. We have parents $P_1$, $P_2$ and $R$, and construction functors $F_1$, $F_2$, such that $P_1 = F_1(R)$ and $P_2 = F_2(R)$. We want to find a new construction functor $F_3$, such that both $P_1$ and $P_2$ coerce into $P_3 = F_3(R)$.

In analogy to a notion of category theory, $P_3$ is called the `pushout()` of $P_1$ and $P_2$; and similarly $F_3$ is called the pushout of $F_1$ and $F_2$.

```
sage: from sage.categories.pushout import pushout
sage: pushout(Fract(ZZ), Poly(ZZ))
Univariate Polynomial Ring in x over Rational Field
```

$F_1 \circ F_2$ and $F_2 \circ F_1$ are natural candidates for the pushout of $F_1$ and $F_2$. However, the order of the functors must rely on a canonical choice. “Indecomposable” construction functors have a `rank`, and this allows to order them canonically:

**Note:** If $F1.rank$ is smaller than $F2.rank$, then the pushout is $F_2 \circ F_1$ (hence, $F_1$ is applied first).
We have

```
sage: Fract.rank, Poly.rank
(5, 9)
```

and thus the pushout is

```
sage: Fract.pushout(Poly), Poly.pushout(Fract)
(Poly[⟨x⟩(FractionField(...)), Poly[⟨x⟩(FractionField(...)))
```

This is why the example above has worked.

However, only “elementary” construction functors have a rank:

```
sage: (Fract*Poly).rank
Traceback (most recent call last):
...
AttributeError: 'CompositeConstructionFunctor' object has no attribute 'rank'
```

## Shuffling composite construction functors

If composed construction functors \( \circ F_2 \circ F_1 \) and \( \circ G_2 \circ G_1 \) are given, then Sage determines their pushout by shuffling the constituents:

- If \( F_1.rank < G_1.rank \) then we apply \( F_1 \) first, and continue with \( \circ F_3 \circ F_2 \) and \( \circ G_2 \circ G_1 \).
- If \( F_1.rank > G_1.rank \) then we apply \( G_1 \) first, and continue with \( \circ F_2 \circ F_1 \) and \( \circ G_3 \circ G_2 \).

If \( F_1.rank == G_1.rank \), then the tie needs to be broken by other techniques (see below).

As an illustration, we first get us some functors and then see how chains of functors are shuffled.

```
sage: AlgClos, R = CC.construction(); AlgClos
AlgebraicClosureFunctor
sage: Compl, R = RR.construction(); Compl
Completion[+Infinity, prec=53]
sage: Matr, R = (MatrixSpace(ZZ,3)).construction(); Matr
MatrixFunctor
sage: AlgClos.rank, Compl.rank, Fract.rank, Poly.rank, Matr.rank
(3, 4, 5, 9, 10)
```

When we apply \( \text{Fract} \cdot \text{AlgClos} \cdot \text{Poly} \cdot \text{AlgClos} \cdot \text{Fract} \) to the ring of integers, we obtain:

```
sage: (Fract*AlgClos*Poly*AlgClos*Fract)(ZZ)
Fraction Field of Univariate Polynomial Ring in x over Algebraic Field
```

When we apply \( \text{Compl} \cdot \text{Matr} \cdot \text{Poly} \) to the ring of integers, we obtain:

```
sage: (Poly*Matr*Compl)(ZZ)
Univariate Polynomial Ring in x over Full MatrixSpace of 3 by 3 dense matrices over Integer Ring
```
Applying the shuffling procedure yields

\[
\begin{align*}
sage: & (\text{Poly}\ast\text{Matr}\ast\text{Fract}\ast\text{Poly}\ast\text{AlgClos}\ast\text{Fract}\ast\text{Compl})(\mathbb{Z}) \\
& \text{Univariate Polynomial Ring in } x \text{ over Full MatrixSpace of } 3 \text{ by } 3 \text{ dense matrices over } \mathbb{Q} \\
& \quad \text{Fraction Field of Univariate Polynomial Ring in } x \text{ over Algebraic Field}
\end{align*}
\]

and this is indeed equal to the pushout found by Sage:

\[
\begin{align*}
sage: & \text{pushout}((\text{Fract}\ast\text{Poly}\ast\text{AlgClos}\ast\text{Fract})(\mathbb{Z}), (\text{Poly}\ast\text{Matr}\ast\text{Compl})(\mathbb{Z})) \\
& \text{Univariate Polynomial Ring in } x \text{ over Full MatrixSpace of } 3 \text{ by } 3 \text{ dense matrices over } \mathbb{Q} \\
& \quad \text{Fraction Field of Univariate Polynomial Ring in } x \text{ over Algebraic Field}
\end{align*}
\]

**Breaking the tie**

If \( F_1.\text{rank} = G_1.\text{rank} \) then Sage’s pushout constructions offers two ways to proceed:

1. Construction functors have a method `merge()` that either returns `None` or returns a construction functor—see below. If either \( F_1.\text{merge}(G_1) \) or \( G_1.\text{merge}(F_1) \) returns a construction functor \( H_1 \), then we apply \( H_1 \) and continue with \( \ldots \circ F_3 \circ F_2 \) and \( \ldots \circ G_3 \circ G_2 \).

2. Construction functors have a method `commutes()`. If either \( F_1.\text{commutes}(G_1) \) or \( G_1.\text{commutes}(F_1) \) returns `True`, then we apply both \( F_1 \) and \( G_1 \) in any order, and continue with \( \ldots \circ F_3 \circ F_2 \) and \( \ldots \circ G_3 \circ G_2 \).

By default, \( F_1.\text{merge}(G_1) \) returns \( F_1 \) if \( F_1 == G_1 \), and returns `None` otherwise. The `commutes()` method exists, but it seems that so far nobody has implemented two functors of the same rank that commute.

**Establishing a default implementation**

The typical application of `merge()` is to provide a coercion between different implementations of the same algebraic structure.

**Note:** If \( F_1(P) \) and \( F_2(P) \) are different implementations of the same thing, then \( F_1.\text{merge}(F_2)(P) \) should return the default implementation.

We want to boldly turn our toy implementation of fraction fields into the new default implementation. Hence:

- Next, we implement a new version of the “usual” fraction field functor, having the same rank, but returning our new implementation.
- We make our new implementation the default, by virtue of a merge method.
- Since our fraction fields accept an optional argument `category`, we pass the optional arguments to the construction functor, which will in turn use it to create a fraction field.

**Warning:**

- Do not override the default `__call__` method of `ConstructionFunctor`—implement `_apply_functor` instead.
- Declare domain and codomain of the functor during initialisation.
sage: from sage.categories.pushout import ConstructionFunctor
sage: class MyFracFunctor(ConstructionFunctor):
    ....:     rank = 5
    ....:     def __init__(self, args=None, kwds=None):
    ....:         self.args = args or ()
    ....:         self.kwds = kwds or {}
    ....:         ConstructionFunctor.__init__(self, IntegralDomains(), Fields())
    ....:     def _apply_functor(self, R):
    ....:         return MyFrac(R,*self.args,**self.kwds)
    ....:     def merge(self, other):
    ....:         if isinstance(other, (type(self), sage.categories.pushout.FractionField)):
    ....:             return self

We verify that our functor can really be used to construct our implementation of fraction fields, and that it can be merged with either itself or the usual fraction field constructor:

sage: MyFracFunctor()
MyFracFunctor

sage: MyFracFunctor()(ZZ)
NewFrac(Integer Ring)

sage: MyFracFunctor().merge(MyFracFunctor())
MyFracFunctor

sage: MyFracFunctor().merge(Fract)
MyFracFunctor

There remains to let our new fraction fields know about the new construction functor. The arguments that were used when creating the fraction field are stored as an attribute—this is a feature provided by CachedRepresentation. We pass all but the first of these arguments to the construction functor, such that the construction functor is able to reconstruct the fraction field.

sage: class MyFrac(MyFrac):
    ....:     def construction(self):
    ....:         return MyFracFunctor(self._reduction[1][1:], self._reduction[2]), self._base()

sage: MyFrac(ZZ['x']).construction()
(MyFracFunctor, Univariate Polynomial Ring in x over Integer Ring)

Due to merging, we have:

sage: pushout(MyFrac(ZZ['x']), Frac(QQ['x']))
NewFrac(Univariate Polynomial Ring in x over Rational Field)
A second note on performance

Being able to do arithmetics involving elements of different parents, with the automatic creation of a pushout to contain the result, is certainly convenient—but one should not rely on it, if speed matters. Simply the conversion of elements into different parents takes time. Moreover, by trac ticket #14058, the pushout may be subject to Python’s cyclic garbage collection. Hence, if one does not keep a strong reference to it, the same parent may be created repeatedly, which is a waste of time. In the following example, we illustrate the slow-down resulting from blindly relying on coercion:

```
sage: ZZxy = ZZ['x','y']
sage: a = ZZxy('x')
sage: b = 1/2
timeit("c = a+b")     # random
10000 loops, best of 3: 172 us per loop
sage: QQxy = QQ['x','y']
timeit("c2 = QQxy(a)+QQxy(b)") # random
10000 loops, best of 3: 168 us per loop
sage: a2 = QQxy(a)
sage: b2 = QQxy(b)
timeit("c2 = a2+b2") # random
100000 loops, best of 3: 10.5 us per loop
```

Hence, if one avoids the explicit or implicit conversion into the pushout, but works in the pushout right away, one can get a more than 10-fold speed-up.

The test suites of the category framework

The category framework does not only provide functionality but also a test framework. This section logically belongs to the section on categories, but without the bits that we have implemented in the section on coercion, our implementation of fraction fields would not have passed the tests yet.

“Abstract” methods

We have already seen above that a category can require/suggest certain parent or element methods, that the user must/should implement. This is in order to smoothly blend with the methods that already exist in Sage.

The methods that ought to be provided are called abstract_method(). Let us see what methods are needed for quotient fields and their elements:

```
sage: from sage.misc.abstract_method import abstract_methods_of_class
sage: abstract_methods_of_class(QuotientFields().parent_class)['optional']
[]
sage: abstract_methods_of_class(QuotientFields().parent_class)['required']
['__contains__']
```

Hence, the only required method (that is actually required for all parents that belong to the category of sets) is an element containment test. That’s fine, because the base class Parent provides a default containment test.

The elements have to provide more:

```
sage: abstract_methods_of_class(QuotientFields().element_class)['optional']
['__add__', '__mul__']
```

(continues on next page)
sage: abstract_methods_of_class(QuotientFields().element_class)['required']
['__bool__', 'denominator', 'numerator']

Hence, the elements must provide denominator() and numerator() methods, and must be able to tell whether they are zero or not. The base class Element provides a default __bool__() method. In addition, the elements may provide Sage’s single underscore arithmetic methods (actually any ring element should provide them).

The _test_... methods

If a parent or element method’s name start with “_test_”, it gives rise to a test in the automatic test suite. For example, it is tested

- whether a parent $P$ actually is an instance of the parent class of the category of $P$,
- whether the user has implemented the required abstract methods,
- whether some defining structural properties (e.g., commutativity) hold.

For example, if one forgets to implement required methods, one obtains the following error:

```
sage: class Foo(Parent):
    ....:     Element = sage.structure.element.Element
    ....:     def __init__(self):
    ....:         Parent.__init__(self, category=QuotientFields())
sage: Bar = Foo()
sage: bar = Bar.element_class(Bar)
sage: bar._test_not_implemented_methods()
Traceback (most recent call last):
  ...
AssertionError: Not implemented method: denominator
```

Here are the tests that form the test suite of quotient fields:

```
sage: [t for t in dir(QuotientFields().parent_class) if t.startswith('_test_')]
['_test_additive_associativity',
'_test_an_element',
'_test_associativity',
'_test_cardinality',
'_test_characteristic',
'_test_characteristic_fields',
'_test_construction',
'_test_distributivity',
'_test_divides',
'_test_elements',
'_test_elements_eq_reflexive',
'_test_elements_eq_symmetric',
'_test_elements_eq_transitive',
'_test_elements_neq',
'_test_euclidean_degree',
'_test_fraction_field',
'_test_gcd_vs_xgcd',
'_test_one',
'_test_prod',
```
We have implemented all abstract methods (or inherit them from base classes), we use the category framework, and we have implemented coercions. So, we are confident that the test suite runs without an error. In fact, it does!

Note: The following trick with the __main__ module is only needed in doctests, not in an interactive session or when defining the classes externally.

```
sage: __main__.MyFrac = MyFrac
sage: __main__.MyElement = MyElement
sage: P = MyFrac(ZZ['x'])
sage: TestSuite(P).run()
```

Let us see what tests are actually performed:

```
sage: TestSuite(P).run(verbosetrue)
running ._test_additive_associativity(). . . pass
running ._test_an_element(). . . pass
running ._test_additive_associativity(). . . pass
running ._test_associativity(). . . pass
running ._test_cardinality(). . . pass
running ._test_category(). . . pass
running ._test_characteristic(). . . pass
running ._test_characteristic_fields(). . . pass
running ._test_construction(). . . pass
running ._test_distributivity(). . . pass
running ._test_elements(). . . pass
running ._test_new(). . . pass
Running the test suite of self.an_element()
running ._test_category(). . . pass
running ._test_new(). . . pass
running ._test_nonzero_equal(). . . pass
running ._test_not_implemented_methods(). . . pass
running ._test_pickling(). . . pass
pass
running ._test_elements_eq_reflexive(). . . pass
running ._test_elements_eq_symmetric(). . . pass
running ._test_elements_eq_transitive(). . . pass
running ._test_elements_neq(). . . pass
running ._test_eq(). . . pass
running ._test_equivalence(). . . pass
running ._test_fraction_field(). . . pass
running ._test_gcd_vs_xgcd(). . . pass
running ._test_new(). . . pass
running ._test_not_implemented_methods(). . . pass
running ._test_one(). . . pass
running ._test_pickling(). . . pass
running ._test_one(). . . pass
running ._test_prod(). . . pass
```
Implementing a new category with additional tests

As one can see, tests are also performed on elements. There are methods that return one element or a list of some elements, relying on “typical” elements that can be found in most algebraic structures.

```
sage: P.an_element(); P.some_elements()
(2):(1)
[(2):(1)]
```

Unfortunately, the list of elements that is returned by the default method is of length one, and that single element could also be a bit more interesting. The method an_element relies on a method _an_element_, so, we implement that. We also override the some_elements method.

```
sage: class MyFrac(MyFrac):
    ....: def _an_element_(self):
    ....:     a = self.base().an_element()
    ....:     b = self.base_ring().an_element()
    ....:     if (a+b)!=0:
    ....:         return self(a)**2/(self(a+b)**3)
    ....:     if b != 0:
    ....:         return self(a)/self(b)**2
    ....:     return self(a)**2*self(b)**3
    ....: def some_elements(self):
    ....:     return [self.an_element(),self(self.base().an_element()),self(self.base_ring().an_element())]
```

```
sage: P = MyFrac(ZZ['x'])
sage: P.an_element(); P.some_elements()
(x^2):(x^3 + 3*x^2 + 3*x + 1)
[(x^2):(x^3 + 3*x^2 + 3*x + 1), (x):(1), (1):(1)]
```

Now, as we have more interesting elements, we may also add a test for the “factor” method. Recall that the method was inherited from the category, but it appears that it is not tested. Normally, a test for a method defined by a category should be provided by the same category. Hence, since factor is defined in the category of quotient fields, a test should be added there. But we won’t change source code here and will instead create a sub--category.

Apparently, If $\epsilon$ is an element of a quotient field, the product of the factors returned by $\epsilon.factor()$ should be equal to $\epsilon$. For forming the product, we use the prod method, that, no surprise, is inherited from another category:

```
sage: P.prod.__module__
'sage.categories.monoids'
```

When we want to create a sub--category, we need to provide a method super_categories(), that returns a list of all immediate super categories (here: category of quotient fields).
Warning: A sub--category $S$ of a category $C$ is not implemented as a sub-class of $C.__class__$. $S$ becomes a sub--category of $C$ only if $S.super_categories()$ returns (a sub--category of) $C$!

The parent and element methods of a category are provided as methods of classes that are the attributes ParentMethods and ElementMethods of the category, as follows:

```python
sage: from sage.categories.category import Category
sage: class QuotientFieldsWithTest(Category):
    # do *not* inherit from QuotientFields, ...
    def super_categories(self):
        return [QuotientFields()]
    # ... declare QuotientFields as a super--category!
    class ParentMethods:
        pass
    class ElementMethods:
        def _test_factorisation(self, **options):
            P = self.parent()
            assert self == P.prod([P(b)**e for b,e in self.factor()])
```

We provide an instance of our quotient field implementation with that new category. Note that categories have a default _repr_ method, that guesses a good string representation from the name of the class: QuotientFieldsWithTest becomes "quotient fields with test".

Note: The following trick with the __main__ module is only needed in doctests, not in an interactive session or when defining the classes externally.

```python
sage: __main__.MyFrac = MyFrac
sage: __main__.MyElement = MyElement
sage: __main__.QuotientFieldsWithTest = QuotientFieldsWithTest
sage: P = MyFrac(ZZ['x'], category=QuotientFieldsWithTest())
```

The new test is inherited from the category. Since an_element() is returning a complicated element, _test_factorisation is a serious test:

```python
sage: P.an_element()._test_factorisation
<bound method QuotientFieldsWithTest.ElementMethods._test_factorisation of (x^2):(x^3 + 3*x^2 + 3*x + 1)>
```

```python
sage: P.an_element().factor()
(x + 1)^-3 * x^2
```

Last, we observe that the new test has automatically become part of the test suite. We remark that the existing tests became more serious as well, since we made sage.structure.parent.Parent.an_element() return something more interesting.

```python
sage: TestSuite(P).run(verbosity=True)
running ._test_additive_associativity() . . . pass
running ._test_an_element() . . . pass
running ._test_associativity() . . . pass
```

(continues on next page)
running ._test_cardinality() . . . pass
running ._test_category() . . . pass
running ._test_characteristic() . . . pass
running ._test_characteristic_fields() . . . pass
running ._test_construction() . . . pass
running ._test_distributivity() . . . pass
running ._test_divides() . . . pass
running ._test_elements() . . .
    Running the test suite of self.an_element()
running ._test_category() . . . pass
running ._test_eq() . . . pass
running ._test_factorisation() . . . pass
running ._test_new() . . . pass
running ._test_nonzero_equal() . . . pass
running ._test_not_implemented_methods() . . . pass
running ._test_pickling() . . . pass
    pass
running ._test_elements_eq_reflexive() . . . pass
running ._test_elements_eq_symmetric() . . . pass
running ._test_elements_eq_transitive() . . . pass
running ._test_neq() . . . pass
running ._test_eq() . . . pass
running ._test_euclidean_degree() . . . pass
running ._test_fraction_field() . . . pass
running ._test_gcd_vs_xgcd() . . . pass
running ._test_new() . . . pass
running ._test_not_implemented_methods() . . . pass
running ._test_one() . . . pass
running ._test_pickling() . . . pass
running ._test_produ() . . . pass
running ._test_quo_rem() . . . pass
running ._test_some_elements() . . . pass
running ._test_zero() . . . pass
running ._test_zero_divisors() . . . pass

Appendix: The complete code

```python
# Importing base classes, ...
import sage
from sage.rings.ring import Field
from sage.structure.element import FieldElement
from sage.categories.category import Category
# ... the UniqueRepresentation tool,
from sage.structure.unique_representation import UniqueRepresentation
# ... some categories, and ...
from sage.categories.fields import Fields
from sage.categories.quotient_fields import QuotientFields
from sage.categories.integral_domains import IntegralDomains
# construction functors
from sage.categories.pushout import ConstructionFunctor
```

(continues on next page)
# Fraction field elements

class MyElement(FieldElement):
    def __init__(self, parent, n, d=None):
        if parent is None:
            raise ValueError("The parent must be provided")
        B = parent.base()
        if d is None:
            # The default denominator is one
            d = B.one()
        # verify that both numerator and denominator belong to the base
        if n not in B or d not in B:
            raise ValueError("Numerator and denominator must be elements of \$%s\$"%B)
        # Numerator and denominator should not just be "in" \$B$,
        # but should be defined as elements of \$B$
        d = B(d)
        n = B(n)
        # the denominator must not be zero
        if d==0:
            raise ZeroDivisionError("The denominator must not be zero")
        # normalize the denominator: WLOG, it shall be non-negative.
        if d<0:
            self.n = -n
            self.d = -d
        else:
            self.n = n
            self.d = d
        FieldElement.__init__(self,parent)

        # Methods required by the category of fraction fields:
        def numerator(self):
            return self.n
        def denominator(self):
            return self.d

        # String representation (single underscore!)
        def __repr__(self):
            return "(%s):(%s)"%(self.n,self.d)

        # Comparison: We can assume that both arguments are coerced
        # into the same parent, which is a fraction field. Hence, we
        # are allowed to use the denominator() and numerator() methods
        # on the second argument.
        def __richcmp__(self, other, op):
            from sage.structure.richcmp import richcmp
            return richcmp(self.n*other.denominator(), other.numerator()*self.d, op)

        # Arithmetic methods, single underscore. We can assume that both
        # arguments are coerced into the same parent.
        # We return instances of self.__class__, because self.__class__ will
        # eventually be a sub-class of MyElement.
        def __add__(self, other):
            (continues on next page)
C = self.__class__
D = self.d*other.denominator()
return C(self.parent(), self.n*other.denominator()+self.d*other.numerator(),D)
def __sub__(self, other):
    C = self.__class__
    D = self.d*other.denominator()
    return C(self.parent(), self.n*other.denominator()-self.d*other.numerator(),D)
def __mul__(self, other):
    C = self.__class__
    return C(self.parent(), self.n*other.numerator(), self.d*other.denominator())
def __div__(self, other):
    C = self.__class__
    return C(self.parent(), self.n*other.denominator(), self.d*other.numerator())

# Inheritance from UniqueRepresentation implements the unique parent
# behaviour. Moreover, it implements pickling (provided that Python
# succeeds to look up the class definition).
class MyFrac(UniqueRepresentation, Field):
    # Implement the category framework for elements, which also
    # makes some basic conversions work.
    Element = MyElement

    # Allow to pass to a different category, by an optional argument
    def __init__(self, base, category=None):
        # Fraction fields only exist for integral domains
        if base not in IntegralDomains():
            raise ValueError("%s is no integral domain" % base)
        # Implement the category framework for the parent
        Field.__init__(self, base, category=category or QuotientFields())

    # Single-underscore method for string representation
    def __repr__(self):
        return "NewFrac(%s)"%repr(self.base())

    # Two methods that are implicitly used in some tests
    def base_ring(self):
        return self.base().base_ring()
    def characteristic(self):
        return self.base().characteristic()

    # Implement conversions. Do not override __call__!
    def _element_constructor_(self, *args, **kwds):
        if len(args)!=1:
            return self.element_class(self, *args, **kwds)
        x = args[0]
        try:
            P = x.parent()
        except AttributeError:
            return self.element_class(self, x, **kwds)
        if P in QuotientFields() and P != self.base():
            return self.element_class(self, x.numerator(), x.denominator(), **kwds)
        return self.element_class(self, x, **kwds)
# Implement coercion from the base and from fraction fields
# over a ring that coerces into the base

def _coerce_map_from_(self, S):
    if self.base().has_coerce_map_from(S):
        return True
    if S in QuotientFields():
        if self.base().has_coerce_map_from(S.base()):
            return True
        if hasattr(S, 'ring_of_integers') and self.base().has_coerce_map_from(S.ring_of_integers()):
            return True

    # Tell how this parent was constructed, in order to enable pushout constructions
    def construction(self):
        return MyFracFunctor(), self.base()

    # return some elements of this parent
    def _an_element_(self):
        a = self.base().an_element()
        b = self.base_ring().an_element()
        if (a+b)!=0:
            return self(a)**2/(self(a+b)**3)
        if b != 0:
            return self(a)/self(b)**2
        return self(a)**2*self(b)**3

    def some_elements(self):
        return [self.an_element(), self(self.base().an_element()), self(self.base_ring().an_element())]

    # A construction functor for our implementation of fraction fields
    class MyFracFunctor(ConstructionFunctor):
        # The rank is the same for Sage's original fraction field functor
        rank = 5
        def __init__(self):
            # The fraction field construction is a functor
            # from the category of integral domains into the category of
            # fields
            # NOTE: We could actually narrow the codomain and use the
            # category QuotientFields()
            ConstructionFunctor.__init__(self, IntegralDomains(), Fields())
            # Applying the functor to an object. Do not override __call__!
            def _apply_functor(self, R):
                return MyFrac(R)
            # Note: To apply the functor to morphisms, implement
            # _apply_functor_to_morphism
            # Make sure that arithmetic involving elements of Frac(R) and
            # MyFrac(R) works and yields elements of MyFrac(R)
            def merge(self, other):
                if isinstance(other, (type(self), sage.categories.pushout.FractionField)):
                    return self

            # (continues on next page)
12.1.17 Tutorial: Implementing Algebraic Structures

*Author: Nicolas M. Thiéry* <nthiery at users.sf.net>, *Jason Bandlow* <jbandlow@gmail.com> *et al.*

This tutorial will cover four concepts:

- endowing free modules and vector spaces with additional algebraic structure
- defining morphisms
- defining coercions and conversions
- implementing algebraic structures with several realizations

At the end of this tutorial, the reader should be able to reimplement by himself the example of algebra with several realizations:

```
sage: Sets().WithRealizations().example()
```

Namely, we consider an algebra $A(S)$ whose basis is indexed by the subsets $s$ of a given set $S$. $A(S)$ is endowed with three natural bases: $F$, $\text{In}$, $\text{Out}$; in the first basis, the product is given by the union of the indexing sets. The $\text{In}$ basis and $\text{Out}$ basis are defined respectively by:

```python
def _test_factorisation(self, **options):
    P = self.parent()
    # The methods prod() and factor() are inherited from
    # some other categories.
    assert self == P.prod([P(b)**e for b,e in self.factor()])
```
\[ I_n = \sum_{t \subseteq s} F_t \quad F_s = \sum_{t \supseteq s} \text{Out}_t \]

Each such basis gives a realization of \( A \), where the elements are represented by their expansion in this basis. In the running exercises we will progressively implement this algebra and its three realizations, with coercions and mixed arithmetic between them.

This tutorial heavily depends on Tutorial: Using free modules and vector spaces. You may also want to read the less specialized thematic tutorial How to implement new algebraic structures.

### Subclassing free modules and including category information

As a warm-up, we implement the group algebra of the additive group \( \mathbb{Z}/5\mathbb{Z} \). Of course this is solely for pedagogical purposes; group algebras are already implemented (see \( \mathbb{Z}\text{Mod}(5)\).\text{algebra}(\mathbb{Z})\)). Recall that a fully functional \( \mathbb{Z} \)-module over this group can be created with the simple command:

```sage
A = CombinatorialFreeModule(ZZ, Zmod(5), prefix='a')
```

We reproduce the same, but by deriving a subclass of \( \text{CombinatorialFreeModule} \):

```sage
class MyCyclicGroupModule(CombinatorialFreeModule):
    def __init__(self, R, n, *args, **kwargs):
        CombinatorialFreeModule.__init__(self, R, Zmod(n), *args, **kwargs)

A = MyCyclicGroupModule(QQ, 6, prefix='a')  # or 4 or 5 or 11 ...
a = A.basis()
A.an_element()
```

We now want to endow \( A \) with its natural product structure, to get the desired group algebra. To define a multiplication, we should be in a category where multiplication makes sense, which is not yet the case:

```sage
A.category()
```

We can look at the available Categories from the documentation in the reference manual or we can use introspection to look through the list of categories to pick one we want:

```sage
E = AlgebrasWithBasis(QQ).example(); E
```

Once we have chosen an appropriate category (here AlgebrasWithBasis), one can look at one example:

```sage
E = AlgebrasWithBasis(QQ).example(); E
```

and browse through its code:

12.1. Thematic tutorial document tree 431
This code is meant as a template for implementing a new algebra. In particular, this template suggests that we need to implement the methods `product_on_basis`, `one_basis`, `_repr_`, and `algebra_generators`. Another way to get this list of methods is to ask the category (TODO: find a slicker idiom for this):

```python
sage: from sage.misc.abstract_method import abstract_methods_of_class
sage: abstract_methods_of_class(AlgebrasWithBasis(QQ).element_class)
{'optional': ['_add_', '_mul_'],
 'required': ['__bool__', 'monomial_coefficients']}
sage: abstract_methods_of_class(AlgebrasWithBasis(QQ).parent_class)
{'optional': ['one_basis', 'product_on_basis'], 'required': ['__contains__']}
```

**Warning:** The result above is not yet necessarily complete; many required methods in the categories are not yet marked as `abstract_methods()`. We also recommend browsing the documentation of this category: `AlgebrasWithBasis`.

Adding these methods, here is the minimal implementation of the group algebra:

```python
sage: class MyCyclicGroupAlgebra(CombinatorialFreeModule):
    ....: def __init__(self, R, n, **keywords):
    ....:     self._group = Zmod(n)
    ....:     CombinatorialFreeModule.__init__(self, R, self._group,
    ....:                                        category=AlgebrasWithBasis(R), **keywords)
    ....: def product_on_basis(self, left, right):
    ....:     return self.monomial( left + right )
    ....: def one_basis(self):
    ....:     return self._group.zero()
    ....: def algebra_generators(self):
    ....:     return Family( [self.monomial( self._group(1) )] )
    ....: def _repr_(self):
    ....:     return "Jason's group algebra of %s over %s"%(self._group, self.base_
    ...
```

Some notes about this implementation:

- Alternatively, we could have defined `product` instead of `product_on_basis`:

  ```python
  ....:     # def product(self, left, right):
  ....:     #     return ## something ##
  ```

- For the sake of readability in this tutorial, we have stripped out all the documentation strings. Of course all of those should be present as in `E`.

- The purpose of `**keywords` is to pass down options like `prefix` to `CombinatorialFreeModules`.

Let us do some calculations:
This seems to work fine, but we would like to put more stress on our implementation to shake potential bugs out of it. To this end, we will use TestSuite, a tool that performs many routine tests on our algebra for us.

Since we defined the class interactively, instead of in a Python module, those tests will complain about “pickling”. We can silence this error by making sage think that the class is defined in a module. We could also just ignore those failing tests for now or call TestSuite with the argument skip = 'est_picking':

```
sage: import __main__
sage: __main__.MyCyclicGroupAlgebra = MyCyclicGroupAlgebra
```

Ok, let’s run the tests:

```
sage: TestSuite(A).run(verbos=True)
running ._test_additive_associativity() ... pass
running ._test_an_element() ... pass
running ._test_associativity() ... pass
running ._test_cardinality() ... pass
running ._test_category() ... pass
running ._test_characteristic() ... pass
running ._test_construction() ... pass
running ._test_distributivity() ... pass
running ._test_elements() ... Running the test suite of self.an_element()
running ._test_category() ... pass
running ._test_eq() ... pass
running ._test_new() ... pass
running ._test_nonzero_equal() ... pass
running ._test_not_implemented_methods() ... pass
running ._test_pickling() ... pass
running ._test_pickling() ... pass
running ._test_elements_eq_reflexive() ... pass
running ._test_elements_eq_symmetric() ... pass
running ._test_elements_eq_transitive() ... pass
running ._test_elements_neq() ... pass
running ._test_eq() ... pass
running ._test_new() ... pass
```
running ._test_notimplemented_methods() ... pass
running ._test_one() ... pass
running ._test_pickling() ... pass
running ._test_prod() ... pass
running ._test_some_elements() ... pass
running ._test_zero() ... pass

For more information on categories, see Elements, parents, and categories in Sage: a primer:

```
sage: sage.categories.primer?
# not tested
```

**Review**

We wanted to implement an algebra, so we:

1. Created the underlying vector space using `CombinatorialFreeModule`
2. Looked at `sage.categories.<tab>` to find an appropriate category
3. Loaded an example of that category, and used `sage.misc.abstract_method.abstract_methods_of_class()`, to see what methods we needed to write
4. Added the category information and other necessary methods to our class
5. Ran `TestSuite` to catch potential discrepancies

**Exercises**

1. Make a tiny modification to `product_on_basis` in “MyCyclicGroupAlgebra” to implement the dual of the group algebra of the cyclic group instead of its group algebra (so the product is now given by \( b_f b_g = \delta_{f,g} b_f \)).

   Run the `TestSuite` tests (you may ignore the “pickling” errors). What do you notice?

   Fix the implementation of one and check that the `TestSuite` tests now pass.

   Add the Hopf algebra structure. Hint: look at the example:

   ```
sage: C = HopfAlgebrasWithBasis(QQ).example()
```

2. Given a set \( S \), say:

   ```
sage: S = Set([1,2,3,4,5])
```

   and a base ring, say:

   ```
sage: R = QQ
```

   implement an \( R \)-algebra:

   ```
sage: F = SubsetAlgebraOnFundamentalBasis(S, R) # todo: not implemented
```

   with a basis \( \{ b_S \}_{S \in \text{Subsets}\, S} \) indexed by the subsets of \( S \):

   ```
sage: Subsets(S)
Subsets of {1, 2, 3, 4, 5}
```
and where the product is defined by $b_i b_j = b_{i+j}$.

### Morphisms

To better understand relationships between algebraic spaces, one wants to consider morphisms between them:

```python
sage: A.module_morphism ?  # not tested
sage: A = MyCyclicGroupAlgebra(QQ, 2, prefix='a')
sage: B = MyCyclicGroupAlgebra(QQ, 6, prefix='b')
sage: A, B
(Jason's group algebra of Ring of integers modulo 2 over Rational Field, Jason's group algebra of Ring of integers modulo 6 over Rational Field)
```

```python
sage: def func_on_basis(g):
    ....:    r""
    ....:    """This function is the 'brain' of a (linear) morphism from A --> B.
    ....:    The input is the index of basis element of the domain (A).
    ....:    The output is an element of the codomain (B).
    ....:    """
    ....:    if g==1:
    ....:        return B.monomial(Zmod(6)(3))  # g==1 in the range A
    ....:    else:
    ....:        return B.one()
```

We can now define a morphism that extends this function to $A$ by linearity:

```python
sage: phi = A.module_morphism(func_on_basis, codomain=B)
sage: f = A.an_element()
sage: f
2*a[0] + 2*a[1]
sage: phi(f)
2*b[0] + 2*b[3]
```

### Exercise

Define a new free module $\mathbf{In}$ with basis indexed by the subsets of $S$, and a morphism $\phi$ from $\mathbf{In}$ to $F$ defined by

$$
\phi(I_n) = \sum_{t \subseteq S} F_t
$$

### Diagonal and Triangular Morphisms

We now illustrate how to specify that a given morphism is diagonal or triangular with respect to some order on the basis, which means that the morphism is invertible and Sage is able to compute the inverse morphism automatically. Currently this feature requires the domain and codomain to have the same index set (in progress ...).

```python
sage: X = CombinatorialFreeModule(QQ, Partitions(), prefix='x'); x = X.basis();
sage: Y = CombinatorialFreeModule(QQ, Partitions(), prefix='y'); y = Y.basis();
```

A diagonal module morphism takes as argument a function whose input is the index of a basis element of the domain, and whose output is the coefficient of the corresponding basis element of the codomain:
sage: def diag_func(p):
...:     if len(p)==0: return 1
...:     else: return p[0]
...:
...:
sage: diag_func(Partition([3,2,1]))
3

sage: X_to_Y = X.module_morphism(diagonal=diag_func, codomain=Y)
sage: f = X.an_element();
sage: f
2*x[] + 2*x[1] + 3*x[2]
sage: X_to_Y(f)

Python fun fact: ~ is the inversion operator (but be careful with int's!):

sage: ~2
1/2
sage: ~(int(2)) # in python this is the bitwise complement: ~x = -x-1
-3

Diagonal module morphisms are invertible:

sage: Y_to_X = ~X_to_Y
sage: f = y[Partition([3])] - 2*y[Partition([2,1])]
sage: f
-2*y[2, 1] + y[3]

sage: Y_to_X(f)
-2*y[2, 1] + 1/3*y[3]

sage: X_to_Y(Y_to_X(f))
-2*y[2, 1] + y[3]

For triangular morphisms, just like ordinary morphisms, we need a function that accepts as input the index of a basis element of the domain and returns an element of the codomain. We think of this function as representing the columns of the matrix of the linear transformation:

sage: def triang_on_basis(p):
...:     return Y.sum_of_monomials(mu for mu in Partitions(sum(p)) if mu >= p)
...:

sage: triang_on_basis([3,2])

sage: X_to_Y = X.module_morphism(triang_on_basis, triangular='lower', unitriangular=True, codomain=Y)
sage: f = x[Partition([[1,1,1]])] + 2*x[Partition([3,2])];
sage: f
x[1, 1, 1] + 2*x[3, 2]

sage: X_to_Y(f)

Triangular module morphisms are also invertible, even if X and Y are both infinite-dimensional:
sage: Y_to_X = ~X_to_Y
sage: f
x[[1, 1, 1]] + 2*x[[3, 2]]
sage: Y_to_X(X_to_Y(f))
x[[1, 1, 1]] + 2*x[[3, 2]]

For details, see `ModulesWithBasis.ParentMethods.module_morphism()` (and also `sage.categories.modules_with_basis.TriangularModuleMorphism`):

```
sage: A.module_morphism
# not tested
```

**Exercise**

Redefine the morphism \( \phi \) from the previous exercise as a morphism that is triangular with respect to inclusion of subsets and define the inverse morphism. You may want to use the following comparison key as `key` argument to `modules_morphism`:

```
sage: def subset_key(s):
    """
    A comparison key on sets that gives a linear extension
    of the inclusion order.
    INPUT:
    - `[1,...] -- set
    EXAMPLES::
    sage: sorted(Subsets([1,2,3]), key=subset_key)
    [[], [1], [2], [3], [1, 2], [1, 3], [2, 3], [1, 2, 3]]
    """
    return (len(s), list(s))
```

**Coercions**

Once we have defined a morphism \( X \to Y \), we can register it as a coercion. This will allow Sage to apply the morphism automatically whenever we combine elements of \( X \) and \( Y \) together. See `http://sagemath.com/doc/reference/coercion.html` for more information. As a training step, let us first define a morphism \( X \to Y \), and register it as a coercion:

```
sage: def triang_on_basis(p):
    """
    \( Y.\sum_{\mu} \mu \) for \( \mu \) in `Partitions(sum(p))` if \( \mu \geq p \)
    \( Y[[3,2]] \) + \( y[[4,1]] + y[[5]] \)
    sage: triang_on_basis([3,2])
y[[3, 2]] + y[[4, 1]] + y[[5]]
sage: X_to_Y = X.module_morphism(triang_on_basis, triangular='lower', unitriangular=True, codomain=Y)
sage: X_to_Y.<tab>  # not tested
sage: X_to_Y.register_as_coercion()
```

Now we can not only convert elements from \( X \) to \( Y \), but we can also do mixed arithmetic with these elements:
Exercise

Use the inverse of $\phi$ to implement the inverse coercion from $F$ to $In$. Reimplement $In$ as an algebra, with a product method making it use $\phi$ and its inverse.

A digression: new bases and quotients of symmetric functions

As an application, we show how to combine what we have learned to implement a new basis and a quotient of the algebra of symmetric functions:

```python
sage: SF = SymmetricFunctions(QQ);  # A graded Hopf algebra
sage: h = SF.homogeneous();          # A particular basis, indexed by partitions (with some additional magic)
```

So, $h$ is a graded algebra whose basis is indexed by partitions. In more detail, $h([i])$ is the sum of all monomials of degree $i$:

```python
sage: h([2]).expand(4)
x0^2 + x0*x1 + x1^2 + x0*x2 + x1*x2 + x2^2 + x0*x3 + x1*x3 + x2*x3 + x3^2
```

and $h(\mu) = \prod (h(p) \text{ for } p \in \mu)$:

```python
sage: h([3,2,2,1]) == h([3]) * h([2]) * h([2]) * h([1])
True
```

Here we define a new basis $(X_\lambda)_\lambda$ by triangularity with respect to $h$; namely, we set $X_\lambda = \sum_{\mu \geq \lambda, |\mu| = |\lambda|} h_\mu$:

```python
sage: class MySFBasis(CombinatorialFreeModule):
    .....:     r""
    .....:     Note: We would typically use SymmetricFunctionAlgebra_generic
    .....:     for this. This is as an example only.
    .....:     """
    .....:
    .....:     def __init__(self, R, *args, **kwargs):
    .....:         """ TODO: Informative doc-string and examples ""
    .....:         CombinatorialFreeModule.__init__(self, R, Partitions(),
    .....:         category=AlgebrasWithBasis(R), *args, **kwargs)
    .....:         self._h = SymmetricFunctions(R).homogeneous()
    .....:         self._to_h = self.module_morphism( self._to_h_on_basis, triangular='lower',
    .....:         unitriangular=True, codomain=self._h)
    .....:         self._from_h = ~(self._to_h)
    .....:         self._to_h.register_as_coercion()
    .....:         self._from_h.register_as_coercion()
    .....:
    .....:     def _to_h_on_basis(self, la):
    .....:         return self._h.sum_of_monomials(mu for mu in Partitions(sum(la)) if mu >=
    .....:         _la)
```

(continues on next page)
....:     def product(self, left, right):
....:         return self( self._h(left) * self._h(right) )
....:     def __repr__(self):
....:         return "Jason's basis for symmetric functions over %s"%self.base_ring()
....:     @cached_method
....:     def one_basis(self):
....:         return Partition([])

sage: X = MySFSubbasis(QQ, prefix='x'); x = X.basis(); h = SymmetricFunctions(QQ)._homogeneous()
sage: f = X(h([2,1,1]) - 2*h([2,2])); # Note the capital X
sage: h(f)
2*h[1, 1, 1] - 2*h[2, 2]

We now implement a quotient of the algebra of symmetric functions obtained by killing any monomial symmetric function $m_{\lambda}$ such that the first part of $\lambda$ is greater than $k$. See Sets.SubcategoryMethods.Subquotients() for more details about implementing quotients:

sage: class MySFQuotient(CombinatorialFreeModule):
....:     r'''The quotient of the ring of symmetric functions by the ideal generated by those monomial symmetric functions whose part is larger than some fixed number ``k``.''
....:     def __init__(self, R, k, prefix=None, *args, **kwargs):
....:         CombinatorialFreeModule.__init__(self, R, Partitions(NonNegativeIntegers(), max_part=k),
....:                         prefix = 'mm',
....:                         category = Algebras(R).Graded().WithBasis().Quotients(), *args,
....:                         **kwargs)
....:         self._k = k
....:         self._m = SymmetricFunctions(R).monomial()
....:         self.lift = self.module_morphism(self._m.monomial)
....:         self.retract = self._m.module_morphism(self._retract_on_basis,
....:                         _codomain=self)
....:         self.lift.register_as_coercion()
....:         self.retract.register_as_coercion()
.. code-block:: python

    def ambient(self):
        return self._m

    def _retract_on_basis(self, mu):
        r""
        Takes the index of a basis element of a monomial symmetric function, and returns the projection of that element to the quotient.
        ""
        if len(mu) > 0 and mu[0] > self._k:
            return self.zero()
        return self.monomial(mu)

    sage: MM = MySFQuotient(QQ, 3)
    sage: mm = MM.basis()
    sage: m = SymmetricFunctions(QQ).monomial()
    sage: P = Partition
    sage: g = m[P([3,2,1])] + 2*m[P([3,3])] + m[P([4,2])]; g
    m[3, 2, 1] + 2*m[3, 3] + m[4, 2]
    sage: f = MM(g); f
    mm[3, 2, 1] + 2*mm[3, 3]
    sage: m(f)
    m[3, 2, 1] + 2*m[3, 3]
    sage: (m(f))^2
    sage: MM( (m(f))^2 - m(f^2) )
    0
Implementing algebraic structures with several realizations

We now return to the subset algebra and use it as an example to show how to implement several different bases for an algebra with automatic coercions between the different bases. We have already implemented three bases for this algebra: the F, In, and Out bases, as well as coercions between them. In real calculations it is convenient to tie these parents together by implementing an object A that models the abstract algebra itself. Then, the parents F, In and Out will be realizations of A, while A will be a parent with realizations. See Sets().WithRealizations for more information about the expected user interface and the rationale.

Here is a brief template highlighting the overall structure:

```python
class MyAlgebra(Parent, UniqueRepresentation):
    def __init__(self, R, ...):
        category = Algebras(R).Commutative()
        Parent.__init__(self, category=category.WithRealizations())
        # attribute initialization, construction of the morphisms
        # between the bases, ...

class Bases(Category_realization_of_parent):
    def super_categories(self):
        A = self.base()
        category = Algebras(A.base_ring()).Commutative()
        return [A.Realizations(), category.Realizations().WithBasis()]

class ParentMethods:
    r"""Code that is common to all bases of the algebra"""

class ElementMethods:
    r"""Code that is common to elements of all bases of the algebra"""

class FirstBasis(CombinatorialFreeModule, BindableClass):
    def __init__(self, A):
        CombinatorialFreeModule.__init__(self, ..., category=A.Bases())
        # implementation of the multiplication, the unit, ...

class SecondBasis(CombinatorialFreeModule, BindableClass):
    def __init__(self, A):
        CombinatorialFreeModule.__init__(self, ..., category=A.Bases())
        # implementation of the multiplication, the unit, ...
```

The class `MyAlgebra` implements a commutative algebra A with several realizations, which we specify in the constructor of `MyAlgebra`. The two bases classes `MyAlgebra.FirstBasis` and `MyAlgebra.SecondBasis` implement different realizations of A that correspond to distinguished bases on which elements are expanded. They are initialized in the category `MyAlgebra.Bases` of all bases of A, whose role is to factor out their common features. In particular, this construction says that they are:

- realizations of A
- realizations of a commutative algebra, with a distinguished basis

**Note:** There is a bit of redundancy here: given that A knows it is a commutative algebra with realizations the infrastructure could, in principle, determine that its realizations are commutative algebras. If this was done then it would be possible to implement `Bases.super_categories` by returning:
However, this has not been implemented yet.

**Note:** Inheriting from BindableCass just provides syntactic sugar: it makes MyAlgebras().FirstBasis() a shorthand for MyAlgebras().FirstBasis(MyAlgebras().FirstBasis()) (binding behavior). The class Bases inherits this binding behavior from Category_realization_of_parent, which is why we can write MyAlgebras().Bases instead of MyAlgebras().Bases(MyAlgebras())

**Note:** More often than not, the constructors for all of the bases will be very similar, if not identical; so we would want to factor it out. Annoyingly, the natural approach of putting the constructor in Bases.ParentMethods does not work because this is an abstract class whereas the constructor handles the concrete implementation of the data structure. Similarly, it would be better if it was only necessary to specify the classes the bases inherit from once, but this can’t code go into Bases for the same reason.

The current recommended solution is to have an additional class Basis that factors out the common concrete features of the different bases:

```python
class Basis(CombinatorialFreeModule, BindableClass):
    def __init__(self, A):
        CombinatorialFreeModule.__init__(self, ..., category=A.Bases())

class FirstBasis(Basis):
    ...

class SecondBasis(Basis):
    ...
```

This solution works but it is not optimal because to share features between the two bases code needs to go into two locations, Basis and Bases, depending on whether they are concrete or abstract, respectively.

We now urge the reader to browse the full code of the following example, which is meant as a complete template for constructing new parents with realizations:

```python
sage: A = Sets().WithRealizations().example(); A
The subset algebra of {1, 2, 3} over Rational Field

sage: A?
# not implemented
```
Review

Congratulations on reading this far!

We have now been through a complete tour of the features needed to implement an algebra with several realizations. The infrastructure for realizations is not tied specifically to algebras; what we have learned applies mutatis mutandis in full generality, for example for implementing groups with several realizations.

12.1.18 How to call a C code (or a compiled library) from Sage?

If you have some C/C++ code that you would like to call from Sage for your own use, this document is for you.

• Do you want to contribute to Sage by adding your interface to its code? The (more complex) instructions are available here.

Calling “hello_world()” from hello.c

Let us suppose that you have a file named ~/my_dir/hello.c containing:

```c
#include <stdio.h>

void hello_world()
{
    printf("Hello World\n");
}
```

In order to call this function from Sage, you must create a Cython file (i.e. a file whose extension is .pyx). Here, ~/my_dir/hello_sage.pyx contains a header describing the signature of the function that you want to call:

```cython
cdef extern from "hello.c":
    void hello_world()

def my_bridge_function():
    hello_world() # This is the C function from hello.c
```

You can now load this file in Sage, and call the C code though the Python function my_bridge_function:

```
sage: %runfile hello_sage.pyx
Compiling ./hello_sage.pyx...
sage: my_bridge_function()
Hello World
```

Arguments and return value

Calling function with more complex arguments and return values works the same way. To learn more about the Cython language, click here

The following example defines a function taking and returning int * pointers, and involves some memory allocation. The C code defines a function whose purpose is to return the sum of two vectors as a third vector.

The C file (double_vector.c)
#include <string.h>

int * sum_of_two_vectors(int n, int * vec1, int * vec2) {
    /*
     * INPUT: two arrays vec1, vec2 of n integers
     * OUTPUT: an array of size n equal to vec1 + vec2
     */
    int * sum = (int *) malloc(n * sizeof(int));
    int i;
    for (i = 0; i < n; i++)
        sum[i] = vec1[i] + vec2[i];
    return sum;
}

The Cython file (double_vector_sage.pyx)

cdef extern from "double_vector.c":
    int * sum_of_two_vectors(int n, int * vec1, int * vec2)

from libc.stdlib cimport malloc, free

def sage_sum_of_vectors(n, list1, list2):
    cdef int * vec1 = <int *> malloc(n * sizeof(int))
    cdef int * vec2 = <int *> malloc(n * sizeof(int))
    # Fill the vectors
    for i in range(n):
        vec1[i] = list1[i]
        vec2[i] = list2[i]
    # Call the C function
    cdef int * vec3 = sum_of_two_vectors(n, vec1, vec2)
    # Save the answer in a Python object
    answer = [vec3[i] for i in range(n)]
    free(vec1)
    free(vec2)
    free(vec3)
    return answer

Call from Sage:

sage: %runfile double_vector_sage.pyx
Compiling ./double_vector_sage.pyx...
sage: sage_sum_of_vectors(3, [1, 1, 1], [2, 3, 4])
[3, 4, 5]
Calling code from a compiled library

The procedure is very similar again. For our purposes, we build a library from the file ~/my_dir/hello.c:

```c
#include <stdio.h>

void hello_world()
{
    printf("Hello World\n");
}
```

We also need a ~/my_dir/hello.h header file:

```c
void hello_world();
```

We can now compile it as a library:

```
[user@localhost ~/my_dir/] gcc -c -Wall -Werror -fpic hello.c
[user@localhost ~/my_dir/] gcc -shared -o libhello.so hello.o
```

The only files that we need now are hello.h and libhello.so (you can remove the others if you like). We must now indicate the location of the .so and .h files in the header of our ~/my_dir/hello_sage.pyx file:

```python
# distutils: libraries = /home/username/my_dir/hello
cdef extern from "hello.h":
    void hello_world()
def my_bridge_function():
    hello_world() # This is the C function from hello.c
```

Note: The instruction # distutils: libraries = /home/username/my_dir/hello indicates that the library is actually named /home/username/my_dir/hello. Change it according to your needs. For more information about these instructions, see http://cython.readthedocs.io/en/latest/src/reference/compilation.html#configuring-the-c-build

We can now load this file in Sage and call the function:

```
sage: %runfile hello_sage.pyx
Compiling ./hello_sage.pyx...
sage: my_bridge_function()
Hello World
```

12.1.19 Numerical Computing with Sage

**Warning:** Beware that this document may be obsolete.

This document is designed to introduce the reader to the tools in Sage that are useful for doing numerical computation. By numerical computation we essentially mean machine precision floating point computations. In particular, things such as optimization, numerical linear algebra, solving ODE's or PDE's numerically, etc.

In the first part of this document the reader is only assumed to be familiar with Python/Sage. In the second section on using compiled code, the computational prerequisites increase and I assume the reader is comfortable with writing
programs in C or Fortran. The third section is on mpi and parallel programming and only requires knowledge of Python, though familiarity with mpi would be helpful.

In the current version of this document the reader is assumed to be familiar with the techniques of numerical analysis. The goal of this document is not to teach you numerical analysis, but to explain how to express your ideas in Sage and Python. Also this document is not meant to be comprehensive. Instead the goal is to be a road map and orient the reader to the packages relevant to numerical computation and to where they can find more information.

**Numerical Tools**

Sage has many different components that may be useful for numerical analysis. In particular three packages deserve mention, they are numpy, SciPy, and cvxopt. Numpy is an excellent package that provides fast array facilities to python. It includes some basic linear algebra routines, vectorized math routines, random number generators, etc. It supports a programming style similar to one would use in matlab and most matlab techniques have an analogue in numpy. SciPy builds on numpy and provides many different packages for optimization, root finding, statistics, linear algebra, interpolation, FFT and dsp tools, etc. Finally cvxopt is an optimization package which can solve linear and quadratic programming problems and also has a nice linear algebra interface. Now we will spend a bit more time on each of these packages.

Before we start let us point out [http://www.scipy.org/NumPy_for_Matlab_Users](http://www.scipy.org/NumPy_for_Matlab_Users), which has a comparison between matlab and numpy and gives numpy equivalents of matlab commands. If you’re not familiar with matlab, thats fine, even better, it means you won’t have any pre-conceived notions of how things should work. Also this [http://www.scipy.org/Wiki/Documentation?action=AttachFile&do=get&target=scipy_tutorial.pdf](http://www.scipy.org/Wiki/Documentation?action=AttachFile&do=get&target=scipy_tutorial.pdf) is a very nice tutorial on SciPy and numpy which is more comprehensive than ours.

### NumPy

NumPy is not imported into sage initially. To use NumPy, you first need to import it.

```sage
sage: import numpy
```

The basic object of computation in NumPy is an array. It is simple to create an array.

```sage
sage: l = numpy.array([1,2,3])
sage: l
array([1, 2, 3])
```

NumPy arrays can store any type of python object. However, for speed, numeric types are automatically converted to native hardware types (i.e., `int`, `float`, etc.) when possible. If the value or precision of a number cannot be handled by a native hardware type, then an array of Sage objects will be created. You can do calculations on these arrays, but they may be slower than using native types. When the numpy array contains Sage or python objects, then the data type is explicitly printed as `object`. If no data type is explicitly shown when NumPy prints the array, the type is either a hardware float or int.

```sage
sage: l = numpy.array([2**40, 3**40, 4**40])
sage: l
array([1099511627776, 12157665459056928801, 1208925819614629174706176], dtype=object)
sage: a = 2.0000000000000000001
sage: a.prec() # higher precision than hardware floating point numbers
67
sage: numpy.array([a,2*a,3*a])
array([2.000000000000000000, 4.000000000000000000, 6.000000000000000000], dtype=object)
```
The `dtype` attribute of an array tells you the type of the array. For fast numerical computations, you generally want this to be some sort of float. If the data type is float, then the array is stored as an array of machine floats, which takes up much less space and which can be operated on much faster.

```python
sage: l = numpy.array([1.0, 2.0, 3.0])
sage: l.dtype
dtype('float64')
```

You can create an array of a specific type by specifying the `dtype` parameter. If you want to make sure that you are dealing with machine floats, it is good to specify `dtype=float` when creating an array.

```python
sage: l = numpy.array([1,2,3], dtype=float)
sage: l.dtype
dtype('float64')
```

You can access elements of a NumPy array just like any list, as well as take slices.

```python
sage: l = numpy.array(range(10), dtype=float)
sage: l[3]
3.0
sage: l[3:6]
array([3., 4., 5.])
```

You can do basic arithmetic operations.

```python
sage: l+l
array([ 0., 2., 4., 6., 8., 10., 12., 14., 16., 18.])
sage: 2.5*l
array([ 0. , 2.5, 5. , 7.5, 10. , 12.5, 15. , 17.5, 20. , 22.5])
```

Note that `l*l` will multiply the elements of `l` componentwise. To get a dot product, use `numpy.dot()`.

```python
sage: l*l
array([ 0., 1., 4., 9., 16., 25., 36., 49., 64., 81.])
sage: numpy.dot(l,l)
285.0
```

We can also create two dimensional arrays.

```python
sage: m = numpy.array([[1,2],[3,4]])
sage: m
array([[ 1, 2],
        [ 3, 4]])
sage: m[1,1]
4
```

This is basically equivalent to the following.

```python
sage: m = numpy.matrix([[1,2],[3,4]])
sage: m
matrix([[ 1, 2],
        [ 3, 4]])
sage: m[0,1]
2
```
The difference is that with `numpy.array()`, \( m \) is treated as just an array of data. In particular \( m \cdot m \) will multiply componentwise, however with `numpy.matrix()`, \( m \cdot m \) will do matrix multiplication. We can also do matrix vector multiplication, and matrix addition

```python
sage: n = numpy.matrix([[1,2],[3,4]],dtype=float)
sage: v = numpy.array([[1],[2]],dtype=float)
sage: n*v
matrix([[ 5.],
        [11.]])
sage: n+n
matrix([[2., 4.],
        [6., 8.]])
```

If \( n \) was created with `numpy.array()`, then to do matrix vector multiplication, you would use `numpy.dot(n,v)`.

All NumPy arrays have a shape attribute. This is a useful attribute to manipulate

```python
sage: n = numpy.array(range(25),dtype=float)
sage: n
array([ 0., 1., 2., 3., 4., 5., 6., 7., 8., 9., 10.,
       22., 23., 24.])
sage: n.shape=(5,5)
sage: n
array([[ 0., 1., 2., 3., 4.],
       [ 5., 6., 7., 8., 9.],
       [10., 11., 12., 13., 14.],
       [15., 16., 17., 18., 19.],
       [20., 21., 22., 23., 24.]])
```

This changes the one-dimensional array into a \( 5 \times 5 \) array.

NumPy arrays can be sliced as well

```python
sage: n = numpy.array(range(25),dtype=float)
sage: n.shape = (5,5)
sage: n[2:4,1:3]
array([[11., 12.],
       [16., 17.]])
```

It is important to note that the sliced matrices are references to the original

```python
sage: m = n[2:4,1:3]  
sage: m[0,0] = 100
sage: n
array([[ 0., 1., 2., 3., 4.],
       [ 5., 6., 7., 8., 9.],
       [10., 100., 12., 13., 14.],
       [15., 16., 17., 18., 19.],
       [20., 21., 22., 23., 24.]])
```

You will note that the original matrix changed. This may or may not be what you want. If you want to change the sliced matrix without changing the original you should make a copy

```python
m=n[2:4,1:3].copy()
```
Some particularly useful commands are

```python
sage: x = numpy.arange(0,2,0.1,dtype=float)
sage: x
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. , 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9])
```

You can see that `numpy.arange()` creates an array of floats increasing by 0.1 from 0 to 2. There is a useful command `numpy.r_()` that is best explained by example

```python
sage: from numpy import r_
sage: j = complex(0,1)
sage: RealNumber = float
sage: Integer = int
sage: n = r_[0.0:5.0]
sage: n
array([0., 1., 2., 3., 4.])
sage: n = r_[0.0:5.0, [0.0]*5]
sage: n
array([0., 1., 2., 3., 4., 0., 0., 0., 0., 0.])
```

`numpy.r_()` provides a shorthand for constructing NumPy arrays efficiently. Note in the above `0.0:5.0` was shorthand for `0.0, 1.0, 2.0, 3.0, 4.0`. Suppose we want to divide the interval from 0 to 5 into 10 intervals. We can do this as follows

```python
sage: r_[0.0:5.0:11*j]
array([0. , 0.5, 1. , 1.5, 2. , 2.5, 3. , 3.5, 4. , 4.5, 5. ])
```

The notation `0.0:5.0:11*j` expands to a list of 11 equally space points between 0 and 5 including both endpoints. Note that `j` is the NumPy imaginary number, but it has this special syntax for creating arrays. We can combine all of these techniques

```python
sage: n = r_[0.0:5.0:11*j,int(5)*[0.0],-5.0:0.0]
sage: n
array([ 0. , 0.5, 1. , 1.5, 2. , 2.5, 3. , 3.5, 4. , 4.5, 5. ,
0. , 0. , 0. , 0. , 0. , -5. , -4. , -3. , -2. , -1. ])
```

Another useful command is `numpy.meshgrid()`, it produces meshed grids. As an example suppose you want to evaluate \( f(x, y) = x^2 + y^2 \) on a an equally spaced grid with \( \Delta x = \Delta y = .25 \) for \( 0 \leq x, y \leq 1 \). You can do that as follows

```python
sage: import numpy
sage: j = complex(0,1)
sage: def f(x,y):
....:   return x**2+y**2
sage: from numpy import meshgrid
sage: x = numpy.r_[0.0:1.0:5*j]
sage: y = numpy.r_[0.0:1.0:5*j]
sage: xx,yy = meshgrid(x,y)
sage: xx
array([[0. , 0.25, 0.5 , 0.75, 1. ],
[0. , 0.25, 0.5 , 0.75, 1. ],
[0. , 0.25, 0.5 , 0.75, 1. ],
[0. , 0.25, 0.5 , 0.75, 1. ]]
```

(continues on next page)
[0. , 0.25, 0.5 , 0.75, 1. ]]

sage: yy
array([[0. , 0. , 0. , 0. , 0. ],
       [0.25, 0.25, 0.25, 0.25, 0.25],
       [0.5 , 0.5 , 0.5 , 0.5 , 0.5 ],
       [0.75, 0.75, 0.75, 0.75, 0.75],
       [1. , 1. , 1. , 1. , 1. ]])

sage: f(xx,yy)
array([[0. , 0.0625, 0.25 , 0.5625, 1. ],
       [0.0625, 0.125 , 0.3125, 0.625 , 1.0625],
       [0.25 , 0.3125, 0.5 , 0.8125, 1.25 ],
       [0.5625, 0.625 , 0.8125, 1.125 , 1.5625],
       [1. , 1.0625, 1.25 , 1.5625, 2. ]])

You can see that numpy.meshgrid() produces a pair of matrices, here denoted \( xx \) and \( yy \), such that \( (xx[i,j], yy[i,j]) \) has coordinates \((x[i], y[j])\). This is useful because to evaluate \( f \) over a grid, we only need to evaluate it on each pair of entries in \( xx, yy \). Since NumPy automatically performs arithmetic operations on arrays componentwise, it is very easy to evaluate functions over a grid with very little code.

A useful module is the numpy.linalg module. If you want to solve an equation \( Ax = b \) do

```python
sage: import numpy
sage: from numpy import linalg
sage: A = numpy.random.randn(5,5)
sage: b = numpy.array(range(1,6))
sage: x = linalg.solve(A,b)
sage: numpy.dot(A,x)
array([1., 2., 3., 4., 5.])
```

This creates a random 5x5 matrix \( A \), and solves \( Ax = b \) where \( b=[0.0,1.0,2.0,3.0,4.0] \). There are many other routines in the numpy.linalg module that are mostly self-explanatory. For example there are qr and lu routines for doing QR and LU decompositions. There is also a command eigs for computing eigenvalues of a matrix. You can always do <function name>? to get the documentation which is quite good for these routines.

Hopefully this gives you a sense of what NumPy is like. You should explore the package as there is quite a bit more functionality.

SciPy

Again I recommend this [http://www.scipy.org/Wiki/Documentation?action=AttachFile&do=get&target=scipy_tutorial.pdf](http://www.scipy.org/Wiki/Documentation?action=AttachFile&do=get&target=scipy_tutorial.pdf). There are many useful SciPy modules, in particular scipy.optimize, scipy.stats, scipy.linalg, scipy.linsolve, scipy.sparse, scipy.integrate, scipy.fftpack, scipy.signal, scipy.special. Most of these have relatively good documentation and often you can figure out what things do from the names of functions. I recommend exploring them. For example if you do

```python
sage: import scipy
sage: from scipy import optimize
```

Then

```python
sage: optimize.[tab]
```

(continued from previous page)
Thematic Tutorials, Release 9.7

will show a list of available functions. You should see a bunch of routines for finding minimum of functions. In particular if you do

```
Sage: optimize.fmin_cg?
```

you find it is a routine that uses the conjugate gradient algorithm to find the minima of a function.

```
Sage: scipy.special.[tab]
```

will show all the special functions that SciPy has. Spending a little bit of time looking around is a good way to familiarize yourself with SciPy. One thing that is sort of annoying, is that often if you do scipy.math:(tab ). You won’t see a module that is importable. For example scipy.math:(tab ) will not show a signal module but

```
Sage: from scipy import signal
```

and then

```
signal.[tab]
```

will show you a large number of functions for signal processing and filter design. All the modules I listed above can be imported even if you can’t see them initially.

**scipy.integrate**

This module has routines related to numerically solving ODE’s and numerical integration. Lets give an example of using an ODE solver. Suppose you want to solve the ode

\[ x''(t) + ux'(t)(x(t)^2 - 1) + x(t) = 0 \]

which as a system reads

\[
\begin{align*}
x' &= y \\
y' &= -x + \mu y(1 - x^2).
\end{align*}
\]

The module we want to use is odeint in scipy.integrate. We can solve this ode, computing the value of \((y, y')\), at 1000 points between 0, and 100 using the following code.

```
Sage: import numpy
Sage: from scipy import integrate
Sage: def f_1(y,t):
....:     return [y[1],-y[0]-10*y[1]**(y[0]**2-1)]
Sage: def j_1(y,t):
....:     return [[0, 1.0],[-2.0*10*y[0]*y[1]-1.0,-10*(y[0]**y[0]-1.0)] ]
Sage: x = numpy.arange(0,100,.1)
Sage: y = integrate.odeint(f_1,[1,0],x,Dfun=j_1)
```

We could plot the solution if we wanted by doing

```
Sage: pts = [(x[i],y[i][0]) for i in range(len(x))]
Sage: point2d(pts).show()
```
Optimization

The Optimization module has routines related to finding roots, least squares fitting, and minimization. (To be Written)

Cvxopt

Cvxopt provides many routines for solving convex optimization problems such as linear and quadratic programming packages. It also has a very nice sparse matrix library that provides an interface to umfpack (the same sparse matrix solver that matlab uses), it also has a nice interface to lapack. For more details on cvxopt please refer to its documentation at http://cvxopt.org/userguide/index.html

Sparse matrices are represented in triplet notation that is as a list of nonzero values, row indices and column indices. This is internally converted to compressed sparse column format. So for example we would enter the matrix

\[
\begin{bmatrix}
2 & 3 & 0 & 0 & 0 \\
3 & 0 & 4 & 0 & 6 \\
0 & -1 & -3 & 2 & 0 \\
0 & 0 & 1 & 0 & 0 \\
0 & 4 & 2 & 0 & 1
\end{bmatrix}
\]

by

```sage
sage: import numpy # optional - cvxopt
sage: from cvxopt.base import spmatrix # optional - cvxopt
sage: from cvxopt.base import matrix as m # optional - cvxopt
sage: from cvxopt import umfpack # optional - cvxopt
sage: Integer = int # optional - cvxopt
sage: V = [2,3, 3,-1,4, 4,-3,1,2, 2, 6,1]
# optional - cvxopt
sage: I = [0,1, 0, 2,4, 1, 2,3,4, 2, 1,4]
# optional - cvxopt
sage: J = [0,0, 1, 1,1, 2, 2,2,2, 3, 4,4]
# optional - cvxopt
sage: A = spmatrix(V,I,J)
# optional - cvxopt
sage: B = numpy.array([1.0]*5)
# optional - cvxopt
sage: B.shape=(5,1)
# optional - cvxopt
sage: print(B)
# optional - cvxopt
[[1.]
 [1.]
 [1.]
 [1.]
 [1.]]
```

To solve an equation \( AX = B \), with \( B = [1,1,1,1,1] \), we could do the following.

```sage
sage: print(A) # optional - cvxopt
[ 2.00e+00  3.00e+00  0   0   0   ]
[ 3.00e+00  0   4.00e+00  0   6.00e+00]
[ 0   -1.00e+00 -3.00e+00  2.00e+00  0   ]
[ 0   0   1.00e+00  0   0   ]
[ 0   4.00e+00  2.00e+00  0   1.00e+00]
```

```sage
sage: C = m(B)
# optional - cvxopt
sage: umfpack.linsolve(A,C)
# optional - cvxopt
```

(continues on next page)
sage: print(C)  # optional - cvxopt
[ 5.79e-01]
[-5.26e-02]
[ 1.00e+00]
[ 1.97e+00]
[-7.89e-01]

Note the solution is stored in $B$ afterward. Also note the m(B), this turns our Numpy array into a format cvxopt understands. You can directly create a cvxopt matrix using cvxopt’s own matrix command, but I personally find Numpy arrays nicer. Also note we explicitly set the shape of the Numpy array to make it clear it was a column vector.

We could compute the approximate minimum degree ordering by doing

```python
sage: RealNumber = float  # optional - cvxopt
sage: Integer = int  # optional - cvxopt
sage: from cvxopt.base import spmatrix  # optional - cvxopt
sage: from cvxopt import amd  # optional - cvxopt
sage: A = spmatrix([10, 3, 5, -2, 5, 2], [0, 2, 1, 2, 2, 3], [0, 0, 1, 1, 2, 3])  # optional - cvxopt
sage: P = amd.order(A)  # optional - cvxopt
sage: print(P)  # optional - cvxopt
[ 1]
[ 0]
[ 2]
[ 3]
```

For a simple linear programming example, if we want to solve

```
minimize  \[-4x_1 - 5x_2\]
subject to  \[2x_1 + x_2 \leq 3\]
            \[x_1 + 2x_2 \leq 3\]
            \[x_1 \geq 0\]
            \[x_2 \geq 0\]
```

```python
sage: RealNumber = float  # optional - cvxopt
sage: Integer = int  # optional - cvxopt
sage: from cvxopt.base import matrix as m  # optional - cvxopt
sage: from cvxopt import solvers  # optional - cvxopt
sage: c = m([-4., -5.])  # optional - cvxopt
sage: G = m([[2., 1., -1., 0.], [1., 2., 0., -1.]])  # optional - cvxopt
sage: h = m([3., 3., 0., 0.])  # optional - cvxopt
sage: sol = solvers.lp(c, G, h)  # random # optional - cvxopt
```

```
p  cost  dcost  gap  pres  dres  k/t
0:  -8.1000e+00  -1.8300e+01  4e+00  0e+00  8e-01  1e+00
1:  -8.8055e+00  -9.4357e+00  2e-01  1e-16  4e-02  3e-02
2:  -8.9981e+00  -9.0049e+00  2e-03  1e-16  5e-04  4e-04
3:  -9.0000e+00  -9.0000e+00  2e-05  3e-16  5e-06  4e-06
4:  -9.0000e+00  -9.0000e+00  2e-07  1e-16  5e-08  4e-08
```

```python
sage: print(sol['x'])  # optional - cvxopt  # ... below since can get -00 or +00_  
˓→depending on architecture
[ 1.00e...00]
[ 1.00e+00]
```
Using Compiled Code Interactively

This section is about using compiled code in Sage. However, since Sage is built on top of Python most of this is valid for Python in general. The exception is that these notes assume you are using Sage’s interface to f2py which makes it more convenient to work with f2py interactively. You should look at the f2py website for information on using the command line f2py tool. The ctypes example will work in any recent Python install. If you are using Sage, then ctypes and f2py are all there already.

Firstly why would we want to write compiled code? Obviously, because its fast, far faster than interpreted Python code. Sage has very powerful facilities that allow one to interactively call compiled code written in C or Fortran. In fact there 2-4 ways to do this depending on exactly what you want to accomplish. One way is to use Cython. Cython is a language that is a hybrid of C and Python based on Pyrex. It has the ability to call external shared object libraries and is very useful for writing Python extension modules. Cython/Pyrex is covered in detail elsewhere in the Sage documentation.

Suppose that you really want to just write Python code, but there is some particularly time intensive piece of your code that you would like to either write in C/Fortran or simply call an external shared library to accomplish. In this case you have three options with varying strengths and weaknesses.

Note that before you try to use compiled code to speed up your bottleneck make sure there isn’t an easier way. In particular, first try to vectorize, that is express your algorithm as arithmetic on vectors or numpy arrays. These arithmetic operations are done directly in C so will be very fast. If your problem does not lend itself to being expressed in a vectorized form then read on.

Before we start let us note that this is in no way a complete introduction to any of the programs we discuss. This is more meant to orient you to what is possible and what the different options will feel like.

f2py

F2py is a very nice package that automatically wraps fortran code and makes it callable from Python. The Fibonacci examples are taken from the f2py webpage http://cens.ioc.ee/projects/f2py2e/.

From the notebook the magic %fortran will automatically compile any fortran code in a cell and all the subroutines will become callable functions (though the names will be converted to lowercase.) As an example paste the following into a cell. It is important that the spacing is correct as by default the code is treated as fixed format fortran and the compiler will complain if things are not in the correct column. To avoid this, you can write fortran 90 code instead by making your first line !f90. There will be an example of this later.

```fortran
%fortran
C FILE: FIB1.F
C
SUBROUTINE FIB(A,N)
C
CALCULATE FIRST N FIBONACCI NUMBERS
C
INTEGER N
REAL*8 A(N)
DO I=1,N
  IF (I.EQ.1) THEN
    A(I) = 0.0D0
  ELSEIF (I.EQ.2) THEN
    A(I) = 1.0D0
  ELSE
    A(I) = A(I-1) + A(I-2)
  ENDIF
ENDDO
```

(continues on next page)
Now evaluate it. It will be automatically compiled and imported into Sage (though the name of imported function will be lowercase). Now we want to try to call it, we need to somehow pass it an array \(A\), and the length of the array \(N\).

The way it works is that numpy arrays will be automatically converted to fortran arrays, and Python scalars converted to fortran scalars. So to call fib we do the following.

```python
import numpy
m=numpy.array([0]*10,dtype=float)
print(m)
fib(m,10)
print(m)
```

Note that fortran is a function that can be called on any string. So if you have a fortran program in a file my prog.f. Then you could do the following

```python
f=open('my_prog.f','r')
s=f.read()
fortran(s)
```

Now all the functions in my prog.f are callable.

It is possible to call external libraries in your fortran code. You simply need to tell f2py to link them in. For example suppose we wish to write a program to solve a linear equation using lapack (a linear algebra library). The function we want to use is called dgesv and it has the following signature.

**SUBROUTINE** DGESV( N, NRHS, A, LDA, IPIV, B, LDB, INFO )

* **N** (input) INTEGER
  * The number of linear equations, i.e., the order of the matrix \(A\). \(N \geq 0\).
*
* **NRHS** (input) INTEGER
  * The number of right hand sides, i.e., the number of columns of the matrix \(B\). \(NRHS \geq 0\).
*
* **A** (input/output) DOUBLE PRECISION array, dimension (LDA,N)
  * On entry, the \(N\)-by-\(N\) coefficient matrix \(A\).
  * On exit, the factors \(L\) and \(U\) from the factorization \(A = P^*L^*U\); the unit diagonal elements of \(L\) are not stored.
*
* **LDA** (input) INTEGER
  * The leading dimension of the array \(A\). \(LDA \geq \text{max}(1,N)\).
*
* **IPIV** (output) INTEGER array, dimension (N)
  * The pivot indices that define the permutation matrix \(P\); row \(i\) of the matrix was interchanged with row \(IPIV(i)\).
*
* **B** (input/output) DOUBLE PRECISION array, dimension (LDB,NRHS)
  * On entry, the \(N\)-by-\(NRHS\) matrix of right hand side matrix \(B\).
  * On exit, if \(INFO = 0\), the \(N\)-by-\(NRHS\) solution matrix \(X\).
we could do the following. Note that the order that library are in the list actually matters as it is the order in which they are passed to gcc. Also fortran.libraries is simply a list of names of libraries that are linked in. You can just directly set this list. So that fortran.libraries=[‘lapack’,’blas’] is equivalent to the following.

```python
fortran.add_library('lapack')
fortran.add_library('blas')
```

Now

```python
%fortran
!f90
Subroutine LinearEquations(A,b,n)
Integer n
Real*8 A(n,n), b(n)
Integer i, j, pivot(n), ok
call DGESV(n, 1, A, n, pivot, b, n, ok)
end
```

There are a couple things to note about this. As we remarked earlier, if the first line of the code is !f90, then it will be treated as fortran 90 code and does not need to be in fixed format. To use the above try

```python
a=numpy.random.randn(10,10)
b=numpy.array(range(10),dtype=float)
x=b.copy()
linearequations(a,x,10)
numpy.dot(a,x)
```

This will solve the linear system ax=b and store the result in b. If your library is not in Sage’s local/lib or in your path you can add it to the search path using

```python
fortran.add_library_path(‘path’).
```

You can also directly set fortran.library paths by assignment. It should be a list of paths (strings) to be passed to gcc. To give you an idea of some more things you can do with f2py, note that using intent statements you can control the way the resulting Python function behaves a bit better. For example consider the following modification of our original fibonacci code.

```c
C FILE: FIB3.F
    SUBROUTINE FIB(A,N)
C
C CALCULATE FIRST N FIBONACCI NUMBERS
C
(continues on next page)
INTEGER N
REAL*8 A(N)
Cf2py intent(in) n
Cf2py intent(out) a
Cf2py depend(n) a
DO I=1,N
   IF (I.EQ.1) THEN
      A(I) = 0.0D0
   ELSEIF (I.EQ.2) THEN
      A(I) = 1.0D0
   ELSE
      A(I) = A(I-1) + A(I-2)
   ENDIF
ENDDO
END
C END FILE FIB3.F

Note the comments with the intent statements. This tells f2py that n is an input parameter and a is the output. This is called as

\[ a=fib(10) \]

In general you will pass everything declared intent(in) to the fortran function and everything declared intent(out) will be returned in a tuple. Note that declaring something intent(in) means you only care about its value before the function is called not afterwards. So in the above n tells us how many fibonacci numbers to compute we need to specify this as an input, however we don’t need to get n back as it doesn’t contain anything new. Similarly A is intent(out) so we don’t need A to have an specific value beforehand, we just care about the contents afterwards. F2py generates a Python function so you only pass those declared intent(in) and supplies empty workspaces for the remaining arguments and it only returns those that are intent(out). All arguments are intent(in) by default.

Consider now the following

\%fortran
  \begin{verbatim}
  Subroutine Rescale(a,b,n)
  Implicit none
  Integer n,i,j
  Real*8 a(n,n), b
  do i = 1,n
     do j=1,n
        a(i,j)=b*a(i,j)
     end do
  end do
  end
  \end{verbatim}

You might be expecting Rescale(a,n) to rescale a numpy matrix a. Alas this doesn’t work. Anything you pass in is unchanged afterwards. Note that in the fibonacci example above, the one dimensional array was changed by the fortran code, similarly the one dimensional vector b was replaced by its solution in the example where we called lapack while the matrix A was not changed even then dgesv says it modifies the input matrix. Why does this not happen with the two dimensional array. Understanding this requires that you are aware of the difference between how fortran and C store arrays. Fortran stores a matrices using column ordering while C stores them using row ordering. That is the matrix

\[
\begin{pmatrix}
  0 & 1 & 2 \\
  3 & 4 & 5
\end{pmatrix}
\]

is stored as

12.1. Thematic tutorial document tree
One-dimensional arrays are stored the same in C and Fortran. Because of this f2py allows the fortran code to operate on one-dimensional vectors in place, so your fortran code will change one-dimensional numpy arrays passed to it. However, since two-dimensional arrays are different by default f2py copies the numpy array (which is stored in C format) into a second array that is in the fortran format (i.e. takes the transpose) and that is what is passed to the fortran function. We will see a way to get around this copying later. First let us point one way of writing the rescale function.

```fortran
Subroutine Rescale(a,b,n)
   Implicit none
   Integer n,i,j
   Real*8 a(n,n), b
   Cf2py intent(in,out) a
   do i = 1,n
      do j=1,n
         a(i,j)=b*a(i,j)
      end do
   end do
end
```

Note that to call this you would use

```
b=rescale(a,2.0).
```

Note here I am not passing in \( n \) which is the dimension of \( a \). Often f2py can figure this out. This is a good time to mention that f2py automatically generates some documentation for the Python version of the function so you can check what you need to pass to it and what it will return. To use this try

```
rescale?
```

The `intent(in,out)` directives tells f2py to take the contents of \( a \) at the end of the subroutine and return them in a numpy array. This still may not be what you want. The original \( a \) that you pass in is unmodified. If you want to modify the original \( a \) that you passed in use `intent(inout)`. This essentially lets your fortran code work with the data inplace.

```fortran
Subroutine Rescale(a,b,n)
   Implicit none
   Integer n,i,j
   Real*8 a(n,n), b
   Cf2py intent(inout) a
   do i = 1,n
      do j=1,n
         a(i,j)=b*a(i,j)
      end do
   end do
end
```

If you wish to have fortran code work with numpy arrays in place, you should make sure that your numpy arrays are stored in fortran’s format. You can ensure this by using the `order=’FORTRAN’` keyword when creating the arrays, as follows.
a = numpy.array([[1, 2], [3, 4]], dtype=float, order='FORTRAN')
rescale(a, 2.0)

After this executes, a will have the rescaled version of itself. There is one final version which combines the previous 
two.

```fortran
Subroutine Rescale(a, b, n)
    Implicit none
    Integer n, i, j
    Real*8 a(n, n), b
Cf2py intent(in, out, overwrite) a
    do i = 1, n
        do j = 1, n
            a(i, j) = b*a(i, j)
        end do
    end do
end
```

The (in, out, overwrite) intent says that if a is in FORTRAN ordering we work in place, however if its not we copy it 
and return the contents afterwards. This is sort of the best of both worlds. Note that if you are repeatedly passing large 
numpy arrays to fortran code, it is very important to avoiding copying the array by using (inout) or (in, out, overwrite). 
Remember though that your numpy array must use Fortran ordering to avoid the copying.

For more examples and more advanced usage of F2py you should refer to the f2py webpage http://cens.ioc.ee/projects/
f2py2e/. The command line f2py tool which is referred to in the f2py documentation can be called from the Sage shell 
using

```bash
!f2py
```

More Interesting Examples with f2py

Let us now look at some more interesting examples using f2py. We will implement a simple iterative method for solving laplace’s equation in a square. Actually, this implementation is taken from 
http://www.scipy.org/PerformancePython?highlight=%28performance%29 page on the scipy website. It has lots of 
information on implementing numerical algorithms in python.

The following fortran code implements a single iteration of a relaxation method for solving Laplace’s equation in a 
square.

```fortran
subroutine timestep(u, n, m, dx, dy, error)
    double precision u(n, m)
    double precision dx, dy, dx2, dy2, dnr_inv, tmp, diff
    integer n, m, i, j
    dx2 = dx*dx
    dy2 = dy*dy
    dnr_inv = 0.5d0 / (dx2 + dy2)
cf2py intent(in) :: dx, dy
cf2py intent(in, out) :: u
cf2py intent(out) :: error
cf2py intent(hide) :: n, m
```

(continues on next page)
error = 0d0
  do 200,j=2,m-1
    do 100,i=2,n-1
      tmp = u(i,j)
      u(i,j) = ((u(i-1,j) + u(i+1,j))*dy2+
        & (u(i,j-1) + u(i,j+1))*dx2)*dnr_inv
      diff = u(i,j) - tmp
      error = error + diff*diff
    100    continue
  200    continue
error = sqrt(error)
end

If you do
timestep?

You find that you need pass timestep a numpy array u, and the grid spacing dx,dy and it will return the updated u, together with an error estimate. To apply this to actually solve a problem use this driver code

```python
import numpy
j=complex(0,1)
num_points=50
u=numpy.zeros((num_points,num_points),dtype=float)
pi_c=float(pi)
x=numpy.r_[0.0:pi_c:num_points*j]
u[0,:]=numpy.sin(x)
u[num_points-1,:]=numpy.sin(x)
def solve_laplace(u,dx,dy):
  iter =0
  err = 2
  while(iter <10000 and err>1e-6):
    (u,err)=timestep(u,dx,dy)
    iter+=1
  return (u,err,iter)
```

Now call the routine using

```python
(sol,err,iter)=solve_laplace(u,pi_c/(num_points-1),pi_c/(num_points-1))
```

This solves the equation with boundary conditions that the right and left edge of the square are half an oscillation of the sine function. With a 51x51 grid that we are using I find that it takes around .2 s to solve this requiring 2750 iterations. If you have the gnuplot package installed (use optional packages() to find its name and install package to install it), then you can plot this using

```python
import Gnuplot
g=Gnuplot.Gnuplot(persist=1)
g('set parametric')
g('set data style lines')
g('set hidden')
g('set contour base')
g('set zrange [-.2:1.2]')
```

(continues on next page)
data=Gnuplot.GridData(sol,x,x,binary=0)
g.splot(data)

To see what we have gained by using f2py let us compare the same algorithm in pure python and a vectorized version using numpy arrays.

```python
def slowTimeStep(u,dx,dy):
    """Takes a time step using straight forward Python loops."""
    nx, ny = u.shape
    dx2, dy2 = dx**2, dy**2
    dnr_inv = 0.5/(dx2 + dy2)
    err = 0.0
    for i in range(1, nx-1):
        for j in range(1, ny-1):
            tmp = u[i,j]
            u[i,j] = ((u[i-1, j] + u[i+1, j])*dy2 +
                       (u[i, j-1] + u[i, j+1])*dx2)*dnr_inv
            diff = u[i,j] - tmp
            err += diff*diff
    return u, numpy.sqrt(err)

def numpyTimeStep(u,dx,dy):
    dx2, dy2 = dx**2, dy**2
    dnr_inv = 0.5/(dx2 + dy2)
    u_old = u.copy()
    # The actual iteration
    u[1:-1, 1:-1] = ((u[0:-2, 1:-1] + u[2:, 1:-1])*dy2 +
                      (u[1:-1,0:-2] + u[1:-1, 2:])*dx2)*dnr_inv
    v = (u - u_old).flat
    return u, numpy.sqrt(numpy.dot(v,v))
```

You can try these out by changing the timestep function used in our driver routine. The python version is slow even on a 50x50 grid. It takes 70 seconds to solve the system in 3000 iterations. It takes the numpy routine 2 seconds to reach the error tolerance in around 5000 iterations. In contrast it takes the f2py routine around .2 seconds to reach the error tolerance using 3000 iterations. I should point out that the numpy routine is not quite the same algorithm since it is a jacobi iteration while the f2py one is gauss-seidel. This is why the numpy version requires more iterations. Even accounting for this you can see the f2py version appears to be around 5 times faster than the numpy version. Actually if you try this on a 500x500 grid I find that it takes the numpy routine 30 seconds to do 500 iterations while it only takes about 2 seconds for the f2py to do this. So the f2py version is really about 15 times faster. On smaller grids each actual iteration is relatively cheap and so the overhead of calling f2py is more evident, on larger examples where the iteration is expensive, the advantage of f2py is clear. Even on the small example it is still very fast. Note that a 500x500 grid in python would take around half an hour to do 500 iterations.

To my knowledge the fastest that you could implement this algorithm in matlab would be to vectorize it exactly like the numpy routine we have. Vector addition in matlab and numpy are comparable. So unless there is some trick I don’t know about, using f2py you can interactively write code 15 times faster than anything you could write in matlab (Please correct me if I’m wrong). You can actually make the f2py version a little bit faster by using intent(in,out,overwrite) and creating the initial numpy array using order='FORTRAN'. This eliminates the unnecessary copying that is occurring in memory.
Ctypes

Ctypes is a very interesting python package which lets you import shared object libraries into python and call them directly. I should say that even though this is called ctypes, it can be used just as well to call functions from libraries written in fortran. The only complication is you need to know what a fortran function looks like to C. This is simple everything is a pointer, so if your fortran function would be called as foo(A,N) where A is an array and N is its length, then to call it from C it takes a pointer to an array of doubles and a pointer to an int. The other thing to be aware of is that from C, fortran functions usually have an underscore appended. That is, a fortran function foo would appear as foo from C (this is usually the case but is compiler dependent). Having said this, the following examples are in C.

As an example suppose you write the following simple C program

```c
#include <stdio.h>

int sum(double *x, int n)
{
    int i;
    double counter;
    counter = 0;
    for(i=0; i<n; i++)
    {
        counter=counter+x[i];
    }
    return counter;
}
```

which you want to call from python. First make a shared object library by doing (at the command line)

```
$ gcc -c sum.c
$ gcc -shared -o sum.so sum.o
```

Note that on OSX -shared should be replaced by -dynamiclib and sum.so should be called sum.dylib Then you can do

```python
from ctypes import *
my_sum=CDLL('sum.so')
a=numpy.array(range(10),dtype=float)
my_sum.sum(a.ctypes.data_as(c_void_p),int(10))
```

Note here that a.ctypes.data_as(c_void_p) returns a ctypes object that is void pointer to the underlying array of a. Note that even though sum takes a double*, as long as we have a pointer to the correct data it doesn’t matter what its type is since it will be automatically cast.

Note that actually there are other ways to pass in the required array of doubles. For example

```python
a=(c_double*10)()
for i in range(10):
    a[i]=i
my_sum.sum(a,int(10))
```

This example only uses ctypes. Ctypes has wrappers for C data types so for example

```python
a=c_double(10.4)
```

would create a ctypes double object which could be passed to a C function. Note that there is a byref function that lets you pass parameters by reference. This is used in the next example. c double*10, is a python object that represents an
array of 10 doubles and

\[
a = (c\_double*10)()\]

sets a equal to an array of 10 doubles. I find this method is usually less useful than using numpy arrays when the data is mathematical as numpy arrays are more well integrated into python and sage.

Here is an example of using ctypes to directly call lapack. Note that this will only work if you have a lapack shared object library on your system. Also on linux the file would be liblapack.so and you will probably use dgesv (OSX use CLAPACK hence the lack of the underscore).

```python
from ctypes import *
def ctypes_solve(m,b,n):
a=CDLL('/usr/lib/liblapack.dylib')
import numpy
p=(c_int*n)()
size=c_int(n)
one=c_int(1)
ok=c_int(0)
a.dgesv(byref(size),byref(one),m.ctypes.data_as(c_void_p),
       byref(size),p,b.ctypes.data_as(c_void_p),byref(size),byref(ok))
```

For completeness, let us consider a way to solve the laplace equation using C types. Suppose you have written a simple solver in C and you want to call it from python so you can easily test different boundary conditions. Your C program might look like this.

```c
#include <math.h>
#include <stdio.h>

double timestep(double *u, int nx, int ny, double dx, double dy)
{
    double tmp, err, diff,dx2,dy2,dnr_inv;
dx2=dx*dx;
    dy2=dy*dy;
dnr_inv=0.5/(dx2+dy2);
    err = 0.0;
    int i,j;

    for (i=1; i<nx-1; ++i) {
        for (j=1; j<ny-1; ++j) {
            tmp = u[i*nx+j];
            u[i*nx+j] = ((u[(i-1)*nx+j] + u[(i+1)*nx+j])*dy2 +
                          (u[i*nx+j-1] + u[i*nx+j+1])*dx2)*dnr_inv;
            diff = u[i*nx+j] - tmp;
            err += diff*diff;
        }
    }

    return sqrt(err);
}

double solve_in_C(double *u, int nx, int ny, double dx, double dy)
{
    double err;
```

(continues on next page)
int iter;
iter = 0;
err = 1;
while(iter <10000 && err > 1e-6)
{
    err=timestep(u,nx,ny,dx,dy);
    iter++;
}
return err;
}

We can compile it by running at the command line

$ gcc -c laplace.c
$ gcc -shared -o laplace.so laplace.o

Now in sage (notebook or command line) execute

from ctypes import *
laplace=CDLL('/home/jkantor/ laplace.so')
laplace.timestep.restype=c_double
laplace.solve_in_C.restype=c_double
import numpy
u=numpy.zeros((51,51),dtype=float)
pi_c=float(pi)
x=numpy.arange(0,pi_c+pi_c/50,pi_c/50,dtype=float)
u[0,:]=numpy.sin(x)
u[50,:]=numpy.sin(x)
def solve(u):
    iter =0
    err = 2
    n=c_int(int(51))
    pi_c= float(pi/50)
    dx= c_double(pi_c)
    while(iter <5000 and err>1e-6):
        err=laplace.timestep(u.ctypes.data_as(c_void_p),n,n,dx,dx)
        iter+=1
        if(iter %50==0):
            print((err,iter))
    return (u,err,iter)

Note the line laplace.timestep.restype=c double. By default ctypes assumes the return values are ints. If they are not you need to tell it by setting restype to the correct return type. If you execute the above code, then solve(u) will solve the system. It is comparable to the fortran solution taking around .2 seconds. Alternatively you could do

n=c_int(int(51))
dx= c_double(float(pi/50))
laplace.solve_in_C(n.ctypes.data_as(c_void_p),n,n,dx,dx)

which computes the solution entirely in C. This is very fast. Admittedly we could have had our fortran routines do the entire solution at the Fortran level and we would have the same speed.
As I said earlier you can just as easily call a shared object library that is written in Fortran using ctypes. The key point is it must be a shared object library and all fortran arguments are passed by reference, that is as pointers or using byref. Also even though we used very simple data types, it is possible to deal with more complicated C structures. For this and more about ctypes see http://python.net/crew/theller/ctypes/

More complicated ctypes example

Here we will look at a more complicated example. First consider the following C code.

```c
#include <stdio.h>
#include <stdlib.h>

struct double_row_element_t {
    double value;
    int col_index;
    struct double_row_element_t * next_element;
};

typedef struct double_row_element_t double_row_element;

typedef struct {
    int nrows;
    int ncols;
    int nnz;
    double_row_element** rows;
} double_sparse_matrix;

double_sparse_matrix * initialize_matrix(int nrows, int ncols)
{
    int i;
    double_sparse_matrix* new_matrix;
    new_matrix = (double_sparse_matrix *) malloc(sizeof(double_sparse_matrix));
    new_matrix->rows = (double_row_element **) malloc(sizeof(double_row_element *)*nrows);
    for(i=0;i<nrows;i++)
    {
        (new_matrix->rows)[i] = (double_row_element *) malloc(sizeof(double_row_element));
        (new_matrix->rows)[i]->value = 0;
        (new_matrix->rows)[i]->col_index = 0;
        (new_matrix->rows)[i]->next_element = 0;
    }
    new_matrix->nrows = nrows;
    new_matrix->ncols = ncols;
    new_matrix->nnz = 0;
    return new_matrix;
}

int free_matrix(double_sparse_matrix * matrix)
{  
(continues on next page)
int i;
double_row_element* next_element;
double_row_element* current_element;
for(i=0;i<matrix->nrows;i++)
{
    current_element = (matrix->rows)[i];
    while(current_element->next_element!=0)
    {
        next_element=current_element->next_element;
        free(current_element);
        current_element=next_element;
    }
    free(current_element);
}
free(matrix->rows);
free(matrix);
return 1;

int set_value(double_sparse_matrix * matrix, int row, int col, double value)
{
    int i;
    i=0;
double_row_element* current_element;
double_row_element* new_element;

    if(row> matrix->nrows || col > matrix->ncols || row <0 || col <0)
        return 1;

    current_element = (matrix->rows)[row];
    while(1)
    {
        if(current_element->col_index==col)
        {
            current_element->value=value;
            return 0;
        }
        else
        if(current_element->next_element!=0)
        {
            if(current_element->next_element->col_index <= col)
                current_element = current_element->next_element;
            else
            if(current_element->next_element->col_index > col)
            {
                new_element = (double_row_element *) malloc(sizeof(double_row_element));
                new_element->value=value;
                new_element->col_index=col;
                new_element->next_element=current_element->next_element;
                current_element->next_element=new_element;
            }
        }
    }
}
```c
    return 0;
}

} else
{
    new_element = (double_row_element *) malloc(sizeof(double_row_element));
    new_element->value=value;
    new_element->col_index=col;
    new_element->next_element=0;
    current_element->next_element=new_element;
    break;
}

}

return 0;
}

double get_value(double_sparse_matrix* matrix, int row, int col)
{
    int i;
    double_row_element * current_element;
    if(row> matrix->nrows || col > matrix->ncols || row <0 || col <0)
        return 0.0;
    current_element = (matrix->rows)[row];
    while(1)
    {
        if(current_element->col_index==col)
        {
            return current_element->value;
        }
        else
        {
            if(current_element->col_index<col && current_element->next_element !=0)
                current_element=current_element->next_element;
            else
                if(current_element->col_index >col || current_element ->next_element==0)
                    return 0;
        }
    }
}
```

Put it in a file called linked_list_sparse.c and compile it using

```
$ gcc -c linked_list_sparse.c
$ gcc -shared -o linked_list_sparse.so linked_list_sparse.o
```

Next consider the following python helper code.
from ctypes import *

class double_row_element(Structure):
    pass

double_row_element._fields_=[("value",c_double),("col_index",c_int),("next_element",
    POINTER(double_row_element))]

class double_sparse_matrix(Structure):
    _fields_=[("nrows",c_int),("ncols",c_int),("nnz",c_int),("rows",
    POINTER(POINTER(double_row_element)))]

double_sparse_pointer=POINTER(double_sparse_matrix)
sparse_library=CDLL("/home/jkantor/linked_list_sparse.so")
initialize_matrix=sparse_library.initialize_matrix
initialize_matrix.restype=double_sparse_pointer
set_value=sparse_library.set_value
get_value=sparse_library.get_value
get_value.restype=c_double
free_matrix=sparse_library.free_matrix

Let's discuss the above code. The original C code stored a sparse matrix as a linked list. The python code uses the
ctypes Structure class to create structures mirroring the structs in the C code. To create python object representing a C
struct, simply create class that derives from Structure. The _fields_ attribute of the class must be set to a list of tuples
of field names and values. Note that in case you need to refer to a struct before it is completely defined (as in the linked
list) you can first declare it with “Pass”, and then specify the field contents as above. Also note the POINTER operator
which creates a pointer out of any ctypes type. We are able to directly call our library as follows.

m=double_sparse_pointer()
m=initialize_matrix(c_int(10),c_int(10))
set_value(m,c_int(4),c_int(4),c_double(5.0))
a=get_value(m,c_int(4),c_int(4))
print("%f"%a)
free_matrix(m)

Note that you can access the contents of a structure just by (struct_object).field name. However for pointers, there is a
contents attribute. So, in the above, m.contents.nrows would let you access the nrows field. In fact you can manually
walk along the linked list as follows.

m=double_sparse_pointer()
m=initialize_matrix(c_int(10),c_int(10))
set_value(m,c_int(4),c_int(4),c_double(5.0))
a=m.contents.rows[4]
b=a.contents.next_element
b.contents.value
free_matrix(m)
Comparison to Cython/Pyrex

It is certainly possible to write a solver in Cython or Pyrex. From the [http://www.scipy.org/PerformancePython?highlight=%28performance%29](http://www.scipy.org/PerformancePython?highlight=%28performance%29) website you can find an example. One potential downside to Cython over the previous solutions is it requires the user to understand how NumPy arrays or Sage matrices are implemented so as to be able to access their internal data. In contrast the scipy, and ctypes examples only require the user to know C or Fortran and from their perspective the NumPy data magically gets passed to C or Fortran with no further thought from them. In order for pyrex to be competitive as a way to interactively write compiled code, the task of accessing the internal structure of NumPy arrays or Sage matrices needs to be hidden.

Parallel Computation

**mpi4py**

MPI which stands for message passing interface is a common library for parallel programming. There is a package mpi4py that builds on the top of mpi, and lets arbitrary python objects be passed between different processes. These packages are not part of the default sage install. To install them do

```
sage: optional_packages()
```

Find the package name openmpi-* and mpi4py-* and do

```
sage: install_package('openmpi-*')
sage: install_package('mpi4py-*')
```

Note that openmpi takes a while to compile (15-20 minutes or so). Openmpi can be run on a cluster, however this requires some set up so that processes on different machines can communicate (though if you are on a cluster this is probably already set up). The simplest case is if you are on a shared memory or multicore system where openmpi will just work with no configuration from you. To be honest, I have never tried to run mpi4py on a cluster, though there is much information about these topics online.

Now, the way that mpi works is you start a group of mpi processes, all of the processes run the same code. Each process has a rank, that is a number that identifies it. The following pseudocode indicates the general format of MPI programs.

```
....
if my rank is n:
   do some computation ...
   send some stuff to the process of rank j
   receive some data from the process of rank k
else if my rank is n+1:
   ....
```

Each processes looks for what it’s supposed to do (specified by its rank) and processes can send data and receive data. Lets give an example. Create a script with the following code in a file mpi_1.py

```
from mpi4py import MPI
comm = MPI.COMM_WORLD
print("hello world")
print("my rank is: %d"%comm.rank)
```

To run it you can do (from the command line in your sage directory)
The command `mpirun -np 5 ./sage -python mpi_1.py` starts 5 copies of a program under mpi. In this case we have 5 copies of sage in pure python mode run the script `mpi_1.py`. The result should be 5 "hello worlds" plus 5 distinct ranks. The two most important mpi operations are sending and receiving. Consider the following example which you should put in a script `mpi_2.py`

```python
from mpi4py import MPI
import numpy
comm = MPI.COMM_WORLD
rank = comm.rank
size = comm.size
v = numpy.array([rank]*5, dtype=float)
comm.send(v, dest=(rank+1)%size)
data = comm.recv(source=(rank-1)%size)
print("my rank is %d"%rank)
print("I received this:")
print(data)
```

The same command as above with `mpi_1.py` replaced by `mpi_2.py` will produce 5 outputs and you will see each process creates an array and then passes it to the next guy (where the last guy passes to the first.) Note that MPI.size is the total number of mpi processes. MPI.COMM_WORLD is the communication world.

There are some subtleties regarding MPI to be aware of. Small sends are buffered. This means if a process sends a small object it will be stored by openmpi and that process will continue its execution and the object it sent will be received whenever the destination executes a receive. However, if an object is large a process will hang until its destination executes a corresponding receive. In fact the above code will hang if `[rank]*5` is replaced by `[rank]*500. It would be better to do

```python
from mpi4py import MPI
import numpy
comm = MPI.COMM_WORLD
rank = comm.rank
size = comm.size
v = numpy.array([rank]*500, dtype=float)
if comm.rank == 0:
    comm.send(v, dest=(rank+1)%size)
if comm.rank > 0:
    data = comm.recv(source=(rank-1)%size)
    comm.send(v, dest=(rank+1)%size)
if comm.rank == 0:
    data = comm.recv(source=size-1)
print("my rank is %d"%rank)
print("I received this:")
print(data)
```

Now the first process initiates a send, and then process 1 will be ready to receive and then he will send and process 2 will be waiting to receive, etc. This will not lock regardless of how large of an array we pass.

A common idiom is to have one process, usually the one with rank 0 act as a leader. That processes sends data out to the other processes and processes the results and decides how further computation should proceed. Consider the following code

```python
from mpi4py import MPI
import numpy
comm = MPI.COMM_WORLD
rank = comm.rank
size = comm.size
```
from mpi4py import MPI
import numpy
sendbuf=[]
root=0
comm = MPI.COMM_WORLD
if comm.rank==0:
    m=numpy.random.randn(comm.size,comm.size)
    print(m)
    sendbuf=m
v=comm.scatter(sendbuf,root)

print("I got this array:")
print(v)

The scatter command takes a list and evenly divides it amongst all the processes. Here the root process creates a matrix (which is viewed as a list of rows) and then scatters it to everybody (roots sendbuf is divided equally amongst the processes). Each process prints the row it got. Note that the scatter command is executed by everyone, but when root executes it, it acts as a send and a receive (root gets one row from itself), while for everyone else it is just a receive.

There is a complementary gather command that collects results from all the processes into a list. The next example uses scatter and gather together. Now the root process scatters the rows of a matrix, each process then squares the elements of the row it gets. Then the rows are all gathered up again by the root process who collects them into a new matrix.

from mpi4py import MPI
import numpy
comm = MPI.COMM_WORLD
sendbuf=[]
root=0
if comm.rank==0:
    m=numpy.array(range(comm.size*comm.size),dtype=float)
    m.shape=(comm.size,comm.size)
    print(m)
    sendbuf=m
v=comm.scatter(sendbuf,root)

print("I got this array:")
print(v)
v=v*v
recvbuf=comm.gather(v,root)
if comm.rank==0:
    print(numpy.array(recvbuf))

There is also a broadcast command that sends a single object to every process. Consider the following small extension. This is the same as before, but now at the end the root process sends everyone the string "done", which is printed out.

v=MPI.COMM_WORLD.scatter(sendbuf,root)
print("I got this array:")
print(v)
v=v*v
recvbuf=MPI.COMM_WORLD.gather(v,root)
if MPI.COMM_WORLD.rank==0:
    print(numpy.array(recvbuf))

(continues on next page)
if MPI.COMM_WORLD.rank==0:
    sendbuf="done"
recvbuf=MPI.COMM_WORLD.bcast(sendbuf,root)
print(recvbuf)

MPI programming is difficult. It is “schizophrenic programming” in that you are writing a single program with
multiple threads of execution “many voices in one head”.

Parallel Laplace Solver

The following code solves Laplace’s equation in parallel on a grid. It divides a square into \( n \) parallel strips where \( n \) is
the number of processes and uses jocobi-iteration. The way the code works is that the root process creates a matrix and
distributes the pieces to the other processes. At each iteration each process passes its upper row to the process above
and its lower row to the process below since they need to know the values at neighboring points to do the iteration.
Then they iterate and repeat. Every 500 iterations the error estimates from the processes are collected using Gather.
you can compare the output of this with the solver we wrote in the section on f2py.

from mpi4py import MPI
import numpy
size=MPI.size
rank=MPI.rank
num_points=500
sendbuf=[]
root=0
dx=1.0/(num_points-1)
from numpy import r_
j=complex(0,1)
rows_per_process=num_points/size
max_iter=5000
num_iter=0
total_err=1
def numpyTimeStep(u,dx,dy):
    dx2, dy2 = dx**2, dy**2
dnr_inv = 0.5/(dx2 + dy2)
u_old=u.copy()
# The actual iteration
u[1:-1, 1:-1] = ((u[0:-2, 1:-1] + u[2:, 1:-1])*dy2 +
               (u[1:-1,0:-2] + u[1:-1, 2:])*dx2)*dnr_inv
v = (u - u_old).flat
return u,numpy.sqrt(numpy.dot(v,v))

if MPI.rank==0:
    print("num_points: %d"%num_points)
    print("dx: %f"%dx)
    print("row_per_process: %d"%rows_per_process)
m=numpy.zeros((num_points,num_points),dtype=float)
pi_c=numpy.pi
x=r_/[0.0:pi_c:num_points*j]
m[0,:]=numpy.sin(x)

(continues on next page)
m[num_points-1,:] = numpy.sin(x)
l = [m[i*rows_per_process:(i+1)*rows_per_process,:] for i in range(size)]
sendbuf = l

my_grid = MPI.COMM_WORLD.Scatter(sendbuf, root)

while num_iter < max_iter and total_err > 10e-6:

    if rank == 0:
        MPI.COMM_WORLD.Send(my_grid[-1,:], 1)

    if rank > 0 and rank < size - 1:
        row_above = MPI.COMM_WORLD.Recv(rank - 1)
        MPI.COMM_WORLD.Send(my_grid[-1,:], rank + 1)

    if rank == size - 1:
        row_above = MPI.COMM_WORLD.Recv(MPI.rank - 1)
        MPI.COMM_WORLD.Send(my_grid[0,:], rank - 1)

    if rank > 0 and rank < size - 1:
        row_below = MPI.COMM_WORLD.Recv(MPI.rank + 1)
        MPI.COMM_WORLD.Send(my_grid[0,:], MPI.rank - 1)

    if rank == 0:
        row_below = MPI.COMM_WORLD.Recv(1)

    if rank > 0 and rank < size - 1:
        row_below.shape = (1, num_points)
        row_above.shape = (1, num_points)
        u, err = numpyTimeStep(r[row_above, my_grid, row_below], dx, dx)
        my_grid = u[1:-1,:]

    if rank == 0:
        row_below.shape = (1, num_points)
        u, err = numpyTimeStep(r[my_grid, row_below], dx, dx)
        my_grid = u[0:-1,:]

    if rank == size - 1:
        row_above.shape = (1, num_points)
        u, err = numpyTimeStep(r[row_above, my_grid], dx, dx)
        my_grid = u[1:,:]

    if num_iter % 500 == 0:
        err_list = MPI.COMM_WORLD.Gather(err, root)
        if rank == 0:
            total_err = 0
            for a in err_list:

(continues on next page)
For small grid sizes, this will be slower than a straightforward serial implementation. This is because there is overhead from the communication, and for small grids the interprocess communication takes more time than just doing the iteration. However, on a 1000x1000 grid I find that using 4 processors, the parallel version takes only 6 seconds while the serial version we wrote earlier takes 20 seconds.

Exercise: Rewrite the above using f2py, so that each process compiles a fortran function and uses that, how fast can you get this?

12.1.20 Three Lectures about Explicit Methods in Number Theory Using Sage

This article is about using the mathematical software Sage to do computations with number fields and modular forms. It was written for the October 2008 Bordeaux meeting on explicit methods in number theory (http://www.math.u-bordeaux.fr/gtem2008/). It assumes no prior knowledge about Sage, but assumes a graduate level background in algebraic number theory.

Introduction

What is Sage?

Sage (see http://sagemath.org) is a comprehensive mathematical software system for computations in many areas of pure and applied mathematics. We program Sage using the mainstream programming language Python (see http://python.org), or its compiled variant Cython. It is also very easy to efficiently use code written in C/C++ from Sage.

The author of this article started the Sage project in 2005.

Sage is free and open source, meaning you can change any part of Sage and redistribute the result without having to pay any license fees, and Sage can also leverage the power of commercial mathematical software such as Magma and Mathematica, if you happen to have access to those closed source commercial systems.

This document assumes no prior knowledge of either Python or Sage. Our goal is to help number theorists do computations involving number fields and modular forms using Sage.

TODO: Overview of Article

As you read this article, please try every example in Sage, and make sure things works as I claim, and do all of the exercises. Moreover, you should experiment by typing in similar examples and checking that the output you get agrees with what you expect.
Using Sage

To use Sage, install it on your computer, and use either the command line or use the notebook by starting Sage as `sage -n`.

We show Sage sessions as follows:

```
sage: factor(123456)
2^6 * 3 * 643
```

This means that if you type `factor(123456)` as input to Sage, then you'll get `2^6 * 3 * 643` as output. If you're using the Sage command line, you type `factor(123456)` and press enter; if you're using the Sage notebook via your web browser, you type `factor(123456)` into an input cell and press shift-enter; in the output cell you'll see `2^6 * 3 * 643`.

After trying the `factor` command in the previous paragraph (do this now!), you should try factoring some other numbers.

**Note:** What happens if you factor a negative number? a rational number?

You can also draw both 2d and 3d pictures using Sage. For example, the following input plots the number of prime divisors of each positive integer up to 500.

```
sage: line([(n, len(factor(n))) for n in [1..500]])
Graphics object consisting of 1 graphics primitive
```

And, this example draws a similar 3d plot:

```
sage: import warnings
sage: warnings.simplefilter('ignore', UserWarning)
sage: v = [[len(factor(n*m)) for n in [1..15]] for m in [1..15]]
sage: list_plot3d(v, interpolation_type='clough')
Graphics3d Object
```

The Sage-Pari-Magma Ecosystem

- **The main difference between Sage and Pari is that Sage is vastly larger than Pari with a much wider range of functionality, and has many more data types and much more structured objects.** Sage in fact includes Pari, and a typical Sage install takes nearly a gigabyte of disk space, whereas a typical Pari install is much more nimble, using only a few megabytes. There are many number-theoretic algorithms that are included in Sage, which have never been implemented in Pari, and Sage has 2d and 3d graphics which can be helpful for visualizing number theoretic ideas, and a graphical user interface. Both Pari and Sage are free and open source, which means anybody can read or change anything in either program, and the software is free.

- **The biggest difference between Sage and Magma is that Magma is closed source, not free, and difficult for users to extend.** This means that most of Magma cannot be changed except by the core Magma developers, since Magma itself is well over two million lines of compiled C code, combined with about a half million lines of interpreted Magma code (that anybody can read and modify). In designing Sage, we carried over some of the excellent design ideas from Magma, such as the parent, element, category hierarchy.

- **Any mathematician who is serious about doing extensive computational work in algebraic number theory and arithmetic geometry is strongly urged to become familiar with all three systems, since they all have their pros and cons.** Pari is sleek and small, Magma has much unique functionality for computations in arithmetic geometry,
and Sage has a wide range of functionality in most areas of mathematics, a large developer community, and much unique new code.

**Number Fields**

**Introduction to Number Fields**

In Sage, we can create the number field \( \mathbb{Q}(\sqrt[3]{2}) \) as follows.

```plaintext
sage: K.<alpha> = NumberField(x^3 - 2)
```

The above creates two Sage objects, \( K \) and \( \alpha \). Here \( K \) “is” (isomorphic to) the number field \( \mathbb{Q}(\sqrt[3]{2}) \), as we confirm below:

```plaintext
sage: K
Number Field in alpha with defining polynomial x^3 - 2
```

and \( \alpha \) is a root of \( x^3 - 2 \), so \( \alpha \) is an abstract choice of \( \sqrt[3]{2} \) (no specific embedding of the number field \( K \) into \( \mathbb{C} \) is chosen by default in Sage-3.1.2):

```plaintext
sage: alpha^3
2
sage: (alpha+1)^3
3*alpha^2 + 3*alpha + 3
```

**The variable \( x \)**

Note that we did not define \( x \) above before using it. You could “break” the above example by redefining \( x \) to be something funny:

```plaintext
sage: x = 1
sage: K.<alpha> = NumberField(x^3 - 2)
Traceback (most recent call last):
  ...TypeError: polynomial (=1) must be a polynomial.
```

The `Traceback` above indicates that there was an error. Potentially lots of detailed information about the error (a “traceback”) may be given after the word `Traceback` and before the last line, which contains the actual error messages.

**Note:** Important: whenever you use Sage and get a big error, look at the last line for the actual error, and only look at the rest if you are feeling adventurous. In the notebook, the part indicated by ... above is not displayed; to see it, click just to the left of the word `Traceback` and the traceback will appear.

If you redefine \( x \) as above, but need to define a number field using the indeterminate \( x \), you have several options. You can reset \( x \) to its default value at the start of Sage, you can redefine \( x \) to be a symbolic variable, or you can define \( x \) to be a polynomial indeterminate (a polygen):

```plaintext
sage: reset('x')
sage: x
x
```

(continues on next page)
Using tab completion to get the methods of an object

One you have created a number field \( K \), type \( K.\) [tab key] to see a list of functions. Type, e.g., \( K.\) minkowski_embedding? [tab key] to see help on the minkowski_embedding command. To see source code, type \( K.\) minkowski_embedding?? [tab key].

Symbolic Expressions

Another natural way for us to create certain number fields is to create a symbolic expression and adjoin it to the rational numbers. Unlike Pari and Magma (and like Mathematica and Maple), Sage also supports manipulation of symbolic expressions and solving equations, without defining abstract structures such as a number fields. For example, we can define a variable \( a = \sqrt{2} \) as an abstract symbolic object by simply typing \( a = \sqrt{2} \). When we type \( \text{parent}(a) \) below, Sage tells us the mathematical object that it views \( a \) as being an element of; in this case, it’s the ring of all symbolic expressions.

\[
\begin{align*}
\text{sage:} & \quad a = \sqrt{2} \\
\text{sage:} & \quad \text{parent}(a) \\
& \quad \text{Symbolic Ring}
\end{align*}
\]

sqrt(2) in Pari and Magma

In particular, typing \( \sqrt{2} \) does not numerically extract an approximation to \( \sqrt{2} \), like it would in Pari or Magma. We illustrate this below by calling Pari (via the gp interpreter) and Magma directly from within Sage. After we evaluate the following two input lines, copies of GP/Pari and Magma are running, and there is a persistent connection between Sage and those sessions.

\[
\begin{align*}
\text{sage:} & \quad \text{gp('sqrt(2)')} \\
& \quad 1.414213562373095048801688724\ldots \\
\text{sage:} & \quad \text{magma('Sqrt(2)')} \\
& \quad 1.414213562373095048801688724\ldots \quad \# \text{ optional - magma}
\end{align*}
\]

You probably noticed a pause when evaluated the second line as Magma started up. Also, note the \# optional comment, which indicates that the line won’t work if you don’t have Magma installed.
Numerically evaluating $\sqrt{2}$

Incidentally, if you want to numerically evaluate $\sqrt{2}$ in Sage, just give the optional `prec` argument to the `sqrt` function, which takes the required number of bits (binary digits) of precision.

```sage
sage: sqrt(2, prec=100)
1.4142135623730950488016887242
```

It’s important to note in computations like this that there is not an *a priori* guarantee that `prec` bits of the *answer* are all correct. Instead, what happens is that Sage creates the number 2 as a floating point number with 100 bits of accuracy, then asks Paul Zimmerman’s MPFR C library to compute the square root of that approximate number.

Arithmetic with $\sqrt{2}$

We return now to our symbolic expression $a = \sqrt{2}$. If you ask to square $a + 1$ you simply get the formal square. To expand out this formal square, we use the `expand` command.

```sage
sage: a = sqrt(2)
sage: (a+1)^2
(sqrt(2) + 1)^2
sage: expand((a+1)^2)
2*sqrt(2) + 3
```

Adjoining a symbolic expression

Given any symbolic expression for which Sage can compute its minimal polynomial, you can construct the number field obtained by adjoining that expression to $\mathbb{Q}$. The notation is quite simple - just type `QQ[a]` where $a$ is the symbolic expression.

```sage
sage: a = sqrt(2)
sage: K.<b> = QQ[a]
sage: K
Number Field in sqrt2 with defining polynomial x^2 - 2 with sqrt2 = 1.414213562373095?
sage: b
sqrt2
sage: (b+1)^2
2*sqrt2 + 3
sage: QQ[a/3 + 5]
Number Field in a with defining polynomial x^2 - 10*x + 223/9 with a = 5.471404520791032?
```

Coercion: `QQ[a]` versus `QQ(a)`

You can’t create the number field $\mathbb{Q}(a)$ in Sage by typing `QQ(a)`, which has a very different meaning in Sage. It means “try to create a rational number from $a$.” Thus `QQ(a)` in Sage is the analogue of `QQ!a` in Magma (Pari has no notion of rings such as $\mathbb{Q}$).

```sage
sage: a = sqrt(2)
sage: QQ(a)
Traceback (most recent call last):
```
...  
TypeError: unable to convert sqrt(2) to a rational

In general, if $X$ is a ring, or vector space or other “parent structure” in Sage, and $a$ is an element, type $X(a)$ to make an element of $X$ from $a$. For example, if $X$ is the finite field of order 7, and $a = 2/5$ is a rational number, then $X(a)$ is the finite field element 6 (as a quick exercise, check that this is mathematically the correct interpretation).

```
sage: X = GF(7); a = 2/5
sage: X(a)
6
```

### Solving a cubic equation

As a slightly less trivial illustration of symbolic manipulation, consider the cubic equation

$$x^3 + \sqrt{2}x + 5 = 0.$$  

In Sage, we can create this equation, and find an exact symbolic solution.

```
sage: x = var('x')
sage: eqn = x^3 + sqrt(2)*x + 5 == 0
sage: a = solve(eqn, x)[0].rhs()
```

The first line above makes sure that the symbolic variable $x$ is defined, the second creates the equation eqn, and the third line solves eqn for $x$, extracts the first solution (there are three), and takes the right hand side of that solution and assigns it to the variable $a$.

### Viewing complicated symbolic expressions

To see the solution nicely typeset, use the `pretty_print` command

```
sage: pretty_print(a)
-\frac{1}{2} (i \sqrt{3} + 1)(\frac{1}{18} \sqrt{8 \sqrt{2} + 675 \sqrt{3} - \frac{5}{2}})^{\frac{1}{3}} + \frac{1}{6} \frac{(-i \sqrt{3} + 1)\sqrt{2}}{(\frac{1}{18} \sqrt{8 \sqrt{2} + 675 \sqrt{3} - \frac{5}{2}})^{\frac{1}{3}}}
```

You can also see the latex needed to paste $a$ into a paper by typing `latex(a)`. The `latex` command works on most Sage objects.

```
sage: latex(a)
-\frac{1}{2} \left( i \sqrt{3} + 1 \right) \left( \frac{1}{18} \sqrt{8 \sqrt{2} + 675 \sqrt{3} - \frac{5}{2}} \right)^{\frac{1}{3}} + \frac{1}{6} \frac{(-i \sqrt{3} + 1)\sqrt{2}}{\left( \frac{1}{18} \sqrt{8 \sqrt{2} + 675 \sqrt{3} - \frac{5}{2}} \right)^{\frac{1}{3}}}
```
Adjoining a root of the cubic

Next, we construct the number field obtained by adjoining the solution $a$ to $\mathbb{Q}$. Notice that the minimal polynomial of the root is $x^6 + 10x^3 - 2x^2 + 25$.

**Warning:** The following tests are currently broken until trac ticket #5338 is fixed.

```
sage: K.<b> = QQ[a]
sage: K
Number Field in a with defining polynomial x^6 + 10*x^3 - 2*x^2 + 25
sage: a.minpoly()
x^6 + 10*x^3 - 2*x^2 + 25
sage: b.minpoly()
x^6 + 10*x^3 - 2*x^2 + 25
```

We can now compute interesting invariants of the number field $K$.

```
sage: K.class_number()
5
sage: K.galois_group().order()
72
```

**Number Fields: Galois Groups and Class Groups**

**Galois Groups**

We can compute the Galois group of a number field using the `galois_group` function, which by default calls Pari (http://pari.math.u-bordeaux.fr/). You do not have to worry about installing Pari, since Pari is part of Sage. In fact, despite appearances much of the difficult algebraic number theory in Sage is actually done by the Pari C library (be sure to also cite Pari in papers that use Sage).

```
sage: K.<alpha> = NumberField(x^6 + 40*x^3 + 1372)
sage: G = K.galois_group()
sage: G
Galois group 6T2 ([3]2) with order 6 of x^6 + 40*x^3 + 1372
```

Internally $G$ is represented as a group of permutations, but we can also apply any element of $G$ to any element of the field:

```
sage: G.order()
6
sage: G.gens()
[(1,2)(3,4)(5,6), (1,4,6)(2,5,3)]
sage: f = G.1; f(alpha)
1/36*alpha^4 + 1/18*alpha
```

Some more advanced number-theoretical tools are available via $G$:
Thematic Tutorials, Release 9.7

`sage: P = K.primes_above(2)[0]
sage: G.inertia_group(P)
Subgroup generated by [(1,4,6)(2,5,3)] of (Galois group 6T2 ([3]2) with order 6 of x^6 +˓→40*x^3 + 1372)
sage: sorted([G.artin_symbol(Q) for Q in K.primes_above(5)])  # random order, see Trac˓→#18308
[(1,3)(2,6)(4,5), (1,2)(3,4)(5,6), (1,5)(2,4)(3,6)]

If the number field is not Galois over \( \mathbb{Q} \), then the `galois_group` command will construct its Galois closure and return the Galois group of that:

`sage: K.<a> = NumberField(x^3 - 2)
sage: G = K.galois_group(names='b'); G
Galois group 3T2 (S3) with order 6 of x^3 - 2
sage: G.order()
6

Some more Galois groups

We compute two more Galois groups of degree 5 extensions, and see that one has Galois group \( S_5 \), so is not solvable by radicals. For these purposes we only need to know the structure of the Galois group as an abstract group, rather than as an explicit group of automorphisms of the splitting field:

`sage: NumberField(x^5 - 2, 'a').galois_group()
Galois group 5T3 (5:4) with order 20 of x^5 - 2
sage: NumberField(x^5 - x + 2, 'a').galois_group()
Galois group 5T5 (S5) with order 120 of x^5 - x + 2

Magma's Galois group command

Recent versions of Magma have an algorithm for computing Galois groups that in theory applies when the input polynomial has any degree. There are no open source implementation of this algorithm (as far as I know). If you have Magma, you can use this algorithm from Sage by calling the `galois_group` function and giving the `algorithm='magma'` option. The return value is one of the groups in the GAP transitive groups database.

`sage: K.<a> = NumberField(x^3 - 2)
sage: K.galois_group(type="gap", algorithm='magma')  # optional - magma
Galois group Transitive group number 2 of degree 3 of the Number Field in a with defining polynomial x^3 - 2

We emphasize that the above example should not work if you don’t have Magma.
Computing complex embeddings

You can also enumerate all complex embeddings of a number field:

```
sage: K.complex_embeddings()
[
    Ring morphism:
      From: Number Field in a with defining polynomial x^3 - 2
      To: Complex Field with 53 bits of precision
      Defn: a |--> -0.629960524947437 - 1.09112363597172*I,
    Ring morphism:
      From: Number Field in a with defining polynomial x^3 - 2
      To: Complex Field with 53 bits of precision
      Defn: a |--> -0.629960524947437 + 1.09112363597172*I,
    Ring morphism:
      From: Number Field in a with defining polynomial x^3 - 2
      To: Complex Field with 53 bits of precision
      Defn: a |--> 1.25992104989487
]
```

Class Numbers and Class Groups

The class group $C_K$ of a number field $K$ is the group of fractional ideals of the maximal order $R$ of $K$ modulo the subgroup of principal fractional ideals. One of the main theorems of algebraic number theory asserts that $C_K$ is a finite group. For example, the quadratic number field $\mathbb{Q} (\sqrt{-23})$ has class number 3, as we see using the Sage class number command.

```
sage: L.<a> = NumberField(x^2 + 23)
sage: L.class_number()
3
```

Quadratic imaginary fields with class number 1

There are only 9 quadratic imaginary field $\mathbb{Q} (\sqrt{D})$ that have class number 1:

$$D = -3, -4, -7, -8, -11, -19, -43, -67, -163$$

To find this list using Sage, we first experiment with making lists in Sage. For example, typing $[1..10]$ makes the list of integers between 1 and 10.

```
sage: [1..10]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

We can also make the list of odd integers between 1 and 11, by typing $[1,3,\ldots,11]$, i.e., by giving the second term in the arithmetic progression.

```
sage: [1,3,\ldots,11]
[1, 3, 5, 7, 9, 11]
```

Applying this idea, we make the list of negative numbers from $-1$ down to $-10$.  

482 Chapter 12. Documentation
Enumerating quadratic imaginary fields with class number 1

The first two lines below makes a list \( v \) of every \( D \) from \(-1\) down to \(-200\) such that \( D \) is a fundamental discriminant (the discriminant of a quadratic imaginary field).

Note: Note that you will not see the ... in the output below; this ... notation just means that part of the output is omitted below.

```
sage: w = [-1,-2,..,-200]
sage: v = [D for D in w if is_fundamental_discriminant(D)]
sage: v
[-3, -4, -7, -8, -11, -15, -19, -20, ..., -195, -199]
```

Finally, we make the list of \( D \) in our list \( v \) such that the quadratic number field \( \mathbb{Q}(\sqrt{D}) \) has class number 1. Notice that \( \text{QuadraticField}(D) \) is a shorthand for \( \text{NumberField}(x^2 - D) \).

```
sage: [D for D in v if QuadraticField(D,'a').class_number()==1]
[-3, -4, -7, -8, -11, -19, -43, -67, -163]
```

Of course, we have not proved that this is the list of all negative \( D \) so that \( \mathbb{Q}(\sqrt{D}) \) has class number 1.

Class number 1 fields

A frustrating open problem is to prove that there are infinitely many number fields with class number 1. It is quite easy to be convinced that this is probably true by computing a bunch of class numbers of real quadratic fields. For example, over 58 percent of the real quadratic number fields with discriminant \( D < 1000 \) have class number 1!

```
sage: w = [1..1000]
sage: v = [D for D in w if is_fundamental_discriminant(D)]
sage: len(v)
302
sage: len([D for D in v if QuadraticField(D,'a').class_number() == 1])
176
sage: 176.0/302
0.582781456953642
```

For more intuition about what is going on, read about the Cohen-Lenstra heuristics.
Class numbers of cyclotomic fields

Sage can also compute class numbers of extensions of higher degree, within reason. Here we use the shorthand `CyclotomicField(n)` to create the number field $\mathbb{Q}(\zeta_n)$.

```sage
sage: CyclotomicField(7)
Cyclotomic Field of order 7 and degree 6
sage: for n in [2..15]:
....:       print("{} {}\n".format(n, CyclotomicField(n).class_number()))
2 1
3 1
...
15 1
```

In the code above, the notation `for n in [2..15]: ...` means “do ... for $n$ equal to each of the integers 2, 3, 4, ..., 15.”

**Note:** Exercise: Compute what is omitted (replaced by ...) in the output of the previous example.

### Assuming conjectures to speed computations

Computations of class numbers and class groups in Sage is done by the Pari C library, and *unlike in Pari,* by default Sage tells Pari *not to assume* any conjectures. This can make some commands vastly slower than they might be directly in Pari, which *does assume unproved conjectures* by default. Fortunately, it is easy to tell Sage to be more permissive and allow Pari to assume conjectures, either just for this one call or henceforth for all number field functions. For example, with `proof=False` it takes only a few seconds to verify, modulo the conjectures assumed by Pari, that the class number of $\mathbb{Q}(\zeta_{23})$ is 3.

```sage
sage: CyclotomicField(23).class_number(proof=False)
3
```

**Note:** Exercise: What is the smallest $n$ such that $\mathbb{Q}(\zeta_n)$ has class number bigger than 1?

### Class group structure

In addition to computing class numbers, Sage can also compute the group structure and generators for class groups. For example, the quadratic field $\mathbb{Q}(\sqrt{-30})$ has class group $C = (\mathbb{Z}/2\mathbb{Z})^\oplus 2$, with generators the ideal classes containing $(5, \sqrt{-30})$ and $(3, \sqrt{-30})$.

```sage
sage: K.<a> = QuadraticField(-30)
sage: C = K.class_group()
sage: C
class group of order 4 with structure C2 x C2 of Number Field in a with defining polynomial x^2 + 30 with a = 5.477225575051661?*I
sage: category(C)
Category of finite enumerated commutative groups
sage: C.gens()
(Fractional ideal class (5, a), Fractional ideal class (3, a))
```
Arithmetic in the class group

In Sage, the notation \(C.i\) means “the \(i^{th}\) generator of the object \(C\),” where the generators are indexed by numbers 0, 1, 2, .... Below, when we write \(C.0 \ast C.1\), this means “the product of the 0th and 1st generators of the class group \(C\).”

```python
sage: K.<a> = QuadraticField(-30)
sage: C = K.class_group()
sage: C.0
Fractional ideal class (5, a)
sage: C.0.ideal()
Fractional ideal (5, a)
sage: I = C.0 * C.1
sage: I
Fractional ideal class (2, a)
```

Next we find that the class of the fractional ideal \((2, \sqrt{-30} + 4/3)\) is equal to the ideal class \(C.0 \ast C.1\).

```python
sage: A = K.ideal([2, a+4/3])
sage: J = C(A)
sage: J
Fractional ideal class (2/3, 1/3*a)
sage: J == C.0*C.1
True
```

Unfortunately, there is currently no Sage function that writes a fractional ideal class in terms of the generators for the class group.

Orders and Relative Extensions

Orders in Number Fields

An order in a number field \(K\) is a subring of \(K\) whose rank over \(\mathbb{Z}\) equals the degree of \(K\). For example, if \(K = \mathbb{Q}(\sqrt{-1})\), then \(\mathbb{Z}[7i]\) is an order in \(K\). A good first exercise is to prove that every element of an order is an algebraic integer.

```python
sage: K.<I> = NumberField(x^2 + 1)
sage: R = K.order(7*I)
sage: R
Order in Number Field in I with defining polynomial x^2 + 1
sage: R.basis()
[1, 7*I]
```

Using the discriminant command, we compute the discriminant of this order

```python
sage: factor(R.discriminant())
-1 * 2^2 * 7^2
```
Constructing the order with given generators

You can give any list of elements of the number field, and it will generate the smallest ring $R$ that contains them.

```
sage: K.<a> = NumberField(x^4 + 2)
sage: K.order([12*a^2, 4*a + 12]).basis()
[1, 4*a, 4*a^2, 16*a^3]
```

If $R$ isn’t of rank equal to the degree of the number field (i.e., $R$ isn’t an order), then you’ll get an error message.

```
sage: K.order([a^2])
Traceback (most recent call last):
... ValueError: the rank of the span of gens is wrong
```

Computing Maximal Orders

We can also compute the maximal order, using the `maximal_order` command, which behind the scenes finds an integral basis using Pari’s `nfbasis` command. For example, $\mathbb{Q}(\sqrt[4]{2})$ has maximal order $\mathbb{Z}[\sqrt[4]{2}]$, and if $\alpha$ is a root of $x^4 + x^2 - 2x + 8$, then $\mathbb{Q}(\alpha)$ has maximal order with $\mathbb{Z}$-basis

$$\frac{1}{2}a^2 + \frac{1}{2}a, a^2.$$

```
sage: K.<a> = NumberField(x^4 + 2)
sage: K.maximal_order().basis()
[1, a, a^2, a^3]
sage: L.<a> = NumberField(x^3 + x^2 - 2*x+8)
sage: L.maximal_order().basis()
[1, 1/2*a^2 + 1/2*a, a^2]
sage: L.maximal_order().basis()[1].minpoly()
x^3 - 2*x^2 + 3*x - 10
```

Functionality for non-maximal orders is minimal

There is still much important functionality for computing with non-maximal orders that is missing in Sage. For example, there is no support at all in Sage for computing with modules over orders or with ideals in non-maximal orders.

```
sage: K.<a> = NumberField(x^3 + 2)
sage: R = K.order(3*a)
sage: R.ideal(5)
Traceback (most recent call last):
... NotImplementedError: ideals of non-maximal orders not yet supported.
```
Relative Extensions

A relative number field \(L\) is a number field of the form \(K(\alpha)\), where \(K\) is a number field, and an absolute number field is a number field presented in the form \(Q(\alpha)\). By the primitive element theorem, any relative number field \(K(\alpha)\) can be written as \(Q(\beta)\) for some \(\beta \in L\). However, in practice it is often convenient to view \(L\) as \(K(\alpha)\). In Symbolic Expressions, we constructed the number field \(Q(\sqrt{2})(\alpha)\), where \(\alpha\) is a root of \(x^3 + \sqrt{2}x + 5\), but not as a relative field—we obtained just the number field defined by a root of \(x^6 + 10x^3 - 2x^2 + 25\).

Constructing a relative number field step by step

To construct this number field as a relative number field, first we let \(K\) be \(Q(\sqrt{2})\).

```
sage: K.<sqrt2> = QuadraticField(2)
```

Next we create the univariate polynomial ring \(R = K[X]\). In Sage, we do this by typing \(R.<X> = K[]\). Here \(R.<X>\) means “create the object \(R\) with generator \(X\)” and \(K[]\) means a “polynomial ring over \(K\)”, where the generator is named based on the aforementioned \(X\) (to create a polynomial ring in two variables \(X, Y\) simply replace \(R.<X>\) by \(R.<X,Y>\)).

```
sage: R.<X> = K[]
sage: R
```

```
Univariate Polynomial Ring in X over Number Field in sqrt2 with defining polynomial x^2 - 2 with sqrt2 = 1.414213562373095?
```

Now we can make a polynomial over the number field \(K = Q(\sqrt{2})\), and construct the extension of \(K\) obtained by adjoining a root of that polynomial to \(K\).

```
sage: L.<a> = K.extension(X^3 + sqrt2*X + 5)
sage: L
```

```
Number Field in a with defining polynomial X^3 + sqrt2*X + 5...
```

Finally, \(L\) is the number field \(Q(\sqrt{2})(\alpha)\), where \(\alpha\) is a root of \(X^3 + \sqrt{2}\alpha + 5\). We can do now do arithmetic in this number field, and of course include \(\sqrt{2}\) in expressions.

```
sage: a^3
-sqrt2*a - 5
sage: a^3 + sqrt2*a
-5
```

Functions on relative number fields

The relative number field \(L\) also has numerous functions, many of which have both relative and absolute version. For example the relative_degree function on \(L\) returns the relative degree of \(L\) over \(K\); the degree of \(L\) over \(Q\) is given by the absolute_degree function. To avoid possible ambiguity degree is not implemented for relative number fields.

```
sage: L.relative_degree()
3
sage: L.absolute_degree()
6
```
**Extra structure on relative number fields**

Given any relative number field you can also an absolute number field that is isomorphic to it. Below we create $M = \mathbb{Q}(b)$, which is isomorphic to $L$, but is an absolute field over $\mathbb{Q}$.

```python
sage: M.<b> = L.absolute_field()
sage: M
Number Field in b with defining polynomial x^6 + 10*x^3 - 2*x^2 + 25
```

The structure function returns isomorphisms in both directions between $M$ and $L$.

```python
sage: M.structure()
(Isomorphism map:
   From: Number Field in b with defining polynomial x^6 + 10*x^3 - 2*x^2 + 25
   To:  Number Field in a with defining polynomial X^3 + sqrt2*X + 5 over its base field,
   \rightarrow
   Isomorphism map:
   From: Number Field in a with defining polynomial X^3 + sqrt2*X + 5 over its base field
   To:  Number Field in b with defining polynomial x^6 + 10*x^3 - 2*x^2 + 25)
```

**Arbitrary towers of relative number fields**

In Sage one can create arbitrary towers of relative number fields (unlike in Pari, where a relative extension must be a single extension of an absolute field).

```python
sage: R.<X> = L[]
sage: Z.<b> = L.extension(X^3 - a)
sage: Z
Number Field in b with defining polynomial X^3 - a over its base field
sage: Z.absolute_degree()
18
```

Note: Exercise: Construct the relative number field $L = K(\sqrt[3]{2} + \sqrt[3]{3})$, where $K = \mathbb{Q}(\sqrt[3]{2}, \sqrt[3]{3})$.

**Relative number field arithmetic can be slow**

One shortcoming with relative extensions in Sage is that behind the scenes all arithmetic is done in terms of a single absolute defining polynomial, and in some cases this can be very slow (much slower than Magma). Perhaps this could be fixed by using Singular’s multivariate polynomials modulo an appropriate ideal, since Singular polynomial arithmetic is extremely fast. Also, Sage has very little direct support for constructive class field theory, which is a major motivation for explicit computation with relative orders; it would be good to expose more of Pari’s functionality in this regard.
A Bird’s Eye View

We now take a whirlwind tour of some of the number theoretical functionality of Sage. There is much that we won’t cover here, but this should help give you a flavor for some of the number theoretic capabilities of Sage, much of which is unique to Sage.

Integer Factorization

Quadratic Sieve

Bill Hart’s quadratic sieve is included with Sage. The quadratic sieve is the best algorithm for factoring numbers of the form \( pq \) up to around 100 digits. It involves searching for relations, solving a linear algebra problem modulo 2, then factoring \( n \) using a relation \( x^2 \equiv y^2 \mod n \).

```sage
qsieve(next_prime(2^90)*next_prime(2^91), time=True) # not tested
([1237940039285380274899124357, 2475880078570760549798248507], '14.94user 0.53system 0:15.72elapsed 98%CPU (0avgtext+0avgdata 0maxresident)k')
```

Using qsieve is twice as fast as Sage’s general factor command in this example. Note that Sage’s general factor command does nothing but call Pari’s factor C library function.

```sage
time factor(next_prime(2^90)*next_prime(2^91)) # not tested
CPU times: user 28.71 s, sys: 0.28 s, total: 28.98 s
Wall time: 29.38 s
1237940039285380274899124357 * 2475880078570760549798248507
```

Obviously, Sage’s factor command should not just call Pari, but nobody has gotten around to rewriting it yet.

GMP-ECM

Paul Zimmermann’s GMP-ECM is included in Sage. The elliptic curve factorization (ECM) algorithm is the best algorithm for factoring numbers of the form \( n = pm \), where \( p \) is not “too big”. ECM is an algorithm due to Hendrik Lenstra, which works by “pretending” that \( n \) is prime, choosing a random elliptic curve over \( \mathbb{Z}/n\mathbb{Z} \), and doing arithmetic on that curve— if something goes wrong when doing arithmetic, we factor \( n \).

In the following example, GMP-ECM is over 10 times faster than Sage’s generic factor function. Again, this emphasizes that Sage’s generic factor command would benefit from a rewrite that uses GMP-ECM and qsieve.

```sage
time ecm.factor(next_prime(2^40) * next_prime(2^300)) # not tested
CPU times: user 0.85 s, sys: 0.01 s, total: 0.86 s
Wall time: 1.73 s
[1099511627791, ...
˓→2037035976334486086268445668409378161051468393665936250636140449354381299763336706183397533]
time factor(next_prime(2^40) * next_prime(2^300)) # not tested
CPU times: user 23.82 s, sys: 0.04 s, total: 23.86 s
Wall time: 24.35 s
1099511627791 *
˓→2037035976334486086268445668409378161051468393665936250636140449354381299763336706183397533]
```
Elliptic Curves

Cremona’s Databases

Cremona’s databases of elliptic curves are part of Sage. The curves up to conductor 10,000 come standard with Sage, and an there is an optional download to gain access to his complete tables. From a shell, you should run

```
sage -i database_cremona_ellcurve
```

to automatically download and install the extended table.

To use the database, just create a curve by giving

```
sage: EllipticCurve('5077a1')
Elliptic Curve defined by y^2 + y = x^3 - 7*x + 6 over Rational Field
```

```
sage: C = CremonaDatabase()
sage: C[37]['allcurves']
{'a1': [[0, 0, 1, -1, 0], 1, 1],
 'b1': [[0, 1, 1, -23, -50], 0, 3],
 'b2': [[0, 1, 1, -1873, -31833], 0, 1],
 'b3': [[0, 1, 1, -3, 1], 0, 3]}
sage: C.isogeny_class('37b')
[Elliptic Curve defined by y^2 + y = x^3 + x^2 - 23*x - 50 over Rational Field, ...]
```

There is also a Stein-Watkins database that contains hundreds of millions of elliptic curves. It’s over a 2GB download though!

Bryan Birch’s Birthday Card

Bryan Birch recently had a birthday conference, and I used Sage to draw the cover of his birthday card by enumerating all optimal elliptic curves of conductor up to 37, then plotting them with thick randomly colored lines. As you can see below, plotting an elliptic curve is as simple as calling the plot method on it. Also, the graphics array command allows us to easily combine numerous plots into a single graphics object.

```
sage: v = cremona_optimal_curves([11..37])
sage: w = [E.plot(thickness=10,rgbcolor=(random(),random(),random())) for E in v]
sage: graphics_array(w, 4, 5).show(axes=False)
```
We can use Sage’s interact feature to draw a plot of an elliptic curve modulo $p$, with a slider that one drags to change the prime $p$. The interact feature of Sage is very helpful for interactively changing parameters and viewing the results. Type interact? for more help and examples and visit the web page http://wiki.sagemath.org/interact.

In the code below we first define the elliptic curve $E$ using the Cremona label 37a. Then we define an interactive function $f$, which is made interactive using the @interact Python decorator. Because the default for $p$ is primes(2,500), the Sage notebook constructs a slider that varies over the primes up to 500. When you drag the slider and let go, a plot is drawn of the affine $\mathbb{F}_p$ points on the curve $E_{\mathbb{F}_p}$. Of course, one should never plot curves over finite fields, which makes this even more fun.

```python
E = EllipticCurve('37a')
@interact
def f(p=primes(2,500)):
    show(plot(E.change_ring(GF(p)),pointsize=30),
         axes=False, frame=True, gridlines="automatic",
         aspect_ratio=1, gridlinesstyle={'rgbcolor':(0.7,0.7,0.7)})
```

Plotting Modulo $p$
Schoof-Elkies-Atkin Point Counting

Sage includes sea.gp, which is a fast implementation of the SEA (Schoof-Elkies-Atkin) algorithm for counting the number of points on an elliptic curve over $\mathbb{F}_p$.

We create the finite field $k = \mathbb{F}_p$, where $p$ is the next prime after $10^{20}$. The next prime command uses Pari’s `nextprime` function, but proves primality of the result (unlike Pari which gives only the next probable prime after a number). Sage also has a next probable prime function.

```python
sage: k = GF(next_prime(10^20))
```

compute its cardinality, which behind the scenes uses SEA.

```python
sage: E = EllipticCurve_from_j(k.random_element())
sage: E.cardinality()  # random, less than a second
99999999999371984255
```
To see how Sage chooses when to use SEA versus other methods, type `E.cardinality??` and read the source code. As of this writing, it simply uses SEA whenever $p > 10^{18}$.

\textbf{$p$-adic Regulators}

Sage has the world’s best code for computing $p$-adic regulators of elliptic curves, thanks to work of David Harvey and Robert Bradshaw. The $p$-adic regulator of an elliptic curve $E$ at a good ordinary prime $p$ is the determinant of the global $p$-adic height pairing matrix on the Mordell-Weil group $E(\mathbb{Q})$. (This has nothing to do with local or Archimedean heights.) This is the analogue of the regulator in the Mazur-Tate-Teitelbaum $p$-adic analogue of the Birch and Swinnerton-Dyer conjecture.

In particular, Sage implements Harvey’s improvement on an algorithm of Mazur-Stein-Tate, which builds on Kiran Kedlaya’s Monsky-Washnitzer approach to computing $p$-adic cohomology groups.

We create the elliptic curve with Cremona label 389a, which is the curve of smallest conductor and rank 2. We then compute both the 5-adic and 997-adic regulators of this curve.

\begin{verbatim}
sage: E = EllipticCurve('389a')
sage: E.padic_regulator(5, 10)
5^2 + 2*5^3 + 2*5^4 + 4*5^5 + 3*5^6 + 4*5^7 + 3*5^8 + 5^9 + O(5^11)
sage: E.padic_regulator(997, 10)
740*997^2 + 916*997^3 + 472*997^4 + 325*997^5 + 697*997^6 + 642*997^7 + 68*997^8 + 860*997^9 + 884*997^10 + O(997^11)
\end{verbatim}

Before the new algorithm mentioned above, even computing a 7-adic regulator to 3 digits of precision was a nontrivial computational challenge. Now in Sage computing the 100003-adic regulator is routine:

\begin{verbatim}
sage: E.padic_regulator(100003, 5)  # a couple of seconds
42582*100003^2 + 35250*100003^3 + 12790*100003^4 + 64078*100003^5 + O(100003^6)
\end{verbatim}

\textbf{$p$-adic $L$-functions}

$p$-adic $L$-functions play a central role in the arithmetic study of elliptic curves. They are $p$-adic analogues of complex analytic $L$-function, and their leading coefficient (at 0) is the analogue of $L^{(r)}(E, 1)/\Omega_E$ in the $p$-adic analogue of the Birch and Swinnerton-Dyer conjecture. They also appear in theorems of Kato, Schneider, and others that prove partial results toward $p$-adic BSD using Iwasawa theory.

The implementation in Sage is mainly due to work of myself, Christian Wuthrich, and Robert Pollack. We use Sage to compute the 5-adic $L$-series of the elliptic curve 389a of rank 2.

\begin{verbatim}
sage: E = EllipticCurve('389a')
sage: L = E.padic_lseries(5)
sage: L
5-adic L-series of Elliptic Curve defined by y^2 + y = x^3 + x^2 - 2*x over Rational Field
sage: L.series(3)
O(5^5) + O(5^2)*T + (4 + 4*5 + O(5^2))*T^2 + (2 + 4*5 + O(5^2))*T^3 + (3 + O(5^2))*T^4 + O(T^5)
\end{verbatim}
Bounding Shafarevich-Tate Groups

Sage implements code to compute numerous explicit bounds on Shafarevich-Tate Groups of elliptic curves. This functionality is only available in Sage, and uses results Kolyvagin, Kato, Perrin-Riou, etc., and unpublished papers of Wuthrich and me.

```python
sage: E = EllipticCurve('11a1')

sage: E.sha().bound()   # so only 2 could divide sha
[2]

sage: E = EllipticCurve('37a1')  # so only 2 could divide sha

sage: E.sha().bound()
([2], 1)

sage: E = EllipticCurve('389a1')

sage: E.sha().bound()
(0, 0)
```

The $(0, 0)$ in the last output above indicates that the Euler systems results of Kolyvagin and Kato give no information about finiteness of the Shafarevich-Tate group of the curve $E$. In fact, it is an open problem to prove this finiteness, since $E$ has rank 2, and finiteness is only known for elliptic curves for which $L(E, 1) \neq 0$ or $L'(E, 1) \neq 0$.

Partial results of Kato, Schneider and others on the $p$-adic analogue of the BSD conjecture yield algorithms for bounding the $p$-part of the Shafarevich-Tate group. These algorithms require as input explicit computation of $p$-adic $L$-functions, $p$-adic regulators, etc., as explained in Stein-Wuthrich. For example, below we use Sage to prove that 5 and 7 do not divide the Shafarevich-Tate group of our rank 2 curve 389a.

```python
sage: E = EllipticCurve('389a1')

sage: sha = E.sha()

sage: sha.p_primary_bound(5)  # iwasawa theory ==> 5 doesn't divide sha
0

sage: sha.p_primary_bound(7)  # iwasawa theory ==> 7 doesn't divide sha
0
```

This is consistent with the Birch and Swinnerton-Dyer conjecture, which predicts that the Shafarevich-Tate group is trivial. Below we compute this predicted order, which is the floating point number $1.000000$ to some precision. That the result is a floating point number helps emphasize that it is an open problem to show that the conjectural order of the Shafarevich-Tate group is even a rational number in general!

```python
sage: E.sha().an()
1.00000000000000
```

Mordell-Weil Groups and Integral Points

Sage includes both Cremona’s mwrank library and Simon’s 2-descent GP scripts for computing Mordell-Weil groups of elliptic curves.

```python
sage: E = EllipticCurve([1,2,5,17,159])

sage: E.conductor()  # not in the Tables
10272987

sage: E.gens()  # a few seconds
[(-3 : 9 : 1), (-3347/3249 : 1873597/185193 : 1)]
```

Sage can also compute the torsion subgroup, isogeny class, determine images of Galois representations, determine reduction types, and includes a full implementation of Tate’s algorithm over number fields.
Sage has the world’s fastest implementation of computation of all integral points on an elliptic curve over \( \mathbb{Q} \), due to work of Cremona, Michael Mardaus, and Tobias Nagel. This is also the only free open source implementation available.

```sage
E = EllipticCurve([1,2,5,7,17])
sage: E.integral_points(both_signs=True)
[(1 : -9 : 1), (1 : 3 : 1)]
```

A very impressive example is the lowest conductor elliptic curve of rank 3, which has 36 integral points.

```sage
E = elliptic_curves.rank(3)[0]
sage: E.integral_points(both_signs=True)  # less than 3 seconds
[(-3 : -1 : 1), (-3 : 0 : 1), (-2 : -4 : 1), (-2 : 3 : 1), ...
 (816 : -23310 : 1), (816 : -23309 : 1)]
```

The algorithm to compute all integral points involves first computing the Mordell-Weil group, then bounding the integral points, and listing all integral points satisfying those bounds. See Cohen’s new GTM 239 for complete details.

```sage
E = elliptic_curves.rank(4)[0]
sage: E.integral_points(both_signs=True)  # about a minute
[(-10 : 3 : 1), (-10 : 7 : 1), ...
 (19405 : -2712802 : 1), (19405 : 2693397 : 1)]
```

**L-functions**

**Evaluation**

We next compute with the complex \( L \)-function

\[
L(E,s) = \prod_{p|\Delta=389} \frac{1}{1 - a_p p^{-s} + p p^{-2s}} \cdot \prod_{p|\Delta=389} \frac{1}{1 - a_p p^{-s}}
\]

of \( E \). Though the above Euler product only defines an analytic function on the right half plane where \( \text{Re}(s) > 3/2 \), a deep theorem of Wiles et al. (the Modularity Theorem) implies that it has an analytic continuation to the whole complex plane and functional equation. We can evaluate the function \( L \) anywhere on the complex plane using Sage (via code of Tim Dokchitser).

```sage
E = EllipticCurve('389a1')
sage: L = E.lseries()
sage: L
Complex L-series of the Elliptic Curve defined by
 y^2 + y = x^3 + x^2 - 2*x over Rational Field
sage: L(1)  #random due to numerical noise
-1.04124792770327e-19
sage: L(1+i)
-0.638409938588039 + 0.715495239204667*I
sage: L(100)
1.00000000000000
```

12.1. Thematic tutorial document tree

495
Taylor Series

We can also compute the Taylor series of $L$ about any point, thanks to Tim Dokchitser’s code.

```
sage: E = EllipticCurve('389a1')
sage: L = E.lseries()
sage: Ld = L.dokchitser()
sage: Ld.taylor_series(1,4)  #random due to numerical noise
-1.28158145691931e-23 + (7.26268290635587e-24)*z + 0.759316500288427*z^2 - 0.430302375833758362*z^3 + O(z^4)
```

GRH

The Generalized Riemann Hypothesis asserts that all nontrivial zeros of $L(E, s)$ are of the form $1 + iy$. Mike Rubinstein has written a C++ program that is part of Sage that can for any $n$ compute the first $n$ values of $y$ such that $1 + iy$ is a zero of $L(E, s)$. It also verifies the Riemann Hypothesis for these zeros (I think). Rubinstein’s program can also do similar computations for a wide class of $L$-functions, though not all of this functionality is as easy to use from Sage as for elliptic curves. Below we compute the first 10 zeros of $L(E, s)$, where $E$ is still the rank 2 curve 389a.

```
sage: L.zeros(10)
[0.000000000, 0.000000000, 2.87609907, 4.41689608, 5.79340263, 6.98596665, 7.47490750, 8.63320525, 9.63307880, 10.3514333]
```

The Matrix of Frobenius on Hyperelliptic Curves

Sage has a highly optimized implementation of the Harvey-Kedaya algorithm for computing the matrix of Frobenius associated to a curve over a finite field. This is an implementation by David Harvey, which is GPL’d and depends only on NTL and zn_poly (a C library in Sage for fast arithmetic in $(\mathbb{Z}/n\mathbb{Z})[x]$).

We import the hypellfrob function and call it on a polynomial over $\mathbb{Z}$.

```
sage: from sage.schemes.hyperelliptic_curves.hypellfrob import hypellfrob
sage: R.<x> = PolynomialRing(ZZ)
sage: f = x^5 + 2*x^2 + x + 1; p = 101
sage: M = hypellfrob(p, 1, f); M
[O(101) O(101) 93 + O(101) 62 + O(101)]
[O(101) O(101) 55 + O(101) 19 + O(101)]
[O(101) O(101) 65 + O(101) 42 + O(101)]
[O(101) O(101) 89 + O(101) 29 + O(101)]
```

We do the same calculation but in $\mathbb{Z}/101^4\mathbb{Z}$, which gives enough precision to recognize the exact characteristic polynomial in $\mathbb{Z}[x]$ of Frobenius as an element of the endomorphism ring. This computation is still very fast, taking only a fraction of a second.

```
sage: M = hypellfrob(p, 4, f)  # about 0.25 seconds
sage: M[0,0]
91844754 + O(101^4)
```

The characteristic polynomial of Frobenius is $x^4 + 7x^3 + 167x^2 + 707x + 10201$, which determines the $\zeta$ function of the curve $y^2 = f(x)$.
Thematic Tutorials, Release 9.7

sage: M.charpoly()
(1 + 0(101^4))*x^4 + (7 + 0(101^4))*x^3 + (167 + 0(101^4))*x^2 + (707 + 0(101^4))*x + 10201 + 0(101^4)

Note: Exercise: Write down zeta function explicitly, count points over some finite fields and see that things match up.

Modular Symbols

Modular symbols play a key role in algorithms for computing with modular forms, special values of \( L \)-functions, elliptic curves, and modular abelian varieties. Sage has the most general implementation of modular symbols available, thanks to work of myself, Jordi Quer (of Barcelona) and Craig Citro (a student of Hida). Moreover, computation with modular symbols is by far my most favorite part of computational mathematics. There is still a lot of tuning and optimization work to be done for modular symbols in Sage, in order for it to be across the board the fastest implementation in the world, since my Magma implementation is still better in some important cases.

Note: I will talk much more about modular symbols in my lecture on Friday, which will be about modular forms and related objects.

We create the space \( M \) of weight 4 modular symbols for a certain congruence subgroup \( \Gamma_H(13) \) of level 13. Then we compute a basis for this space, expressed in terms of Manin symbols. Finally, we compute the Hecke operator \( T_2 \) acting on \( M \), find its characteristic polynomial and factor it. We also compute the dimension of the cuspidal subspace.

\[
\begin{align*}
sage: & M = ModularSymbols(GammaH(13,[3]), weight=4) \\
sage: & M \\
Modular Symbols space of dimension 14 for Congruence Subgroup Gamma_H(13) with H generated by [3] of weight 4 with sign 0 over Rational Field \\
sage: & M.basis() \\
([X^2,(0,4)], [X^2,(0,7)], [X^2,(4,10)], [X^2,(4,11)], [X^2,(4,12)], [X^2,(7,3)], [X^2,(7,5)], [X^2,(7,6)], [X^2,(7,7)], [X^2,(7,8)], [X^2,(7,9)], [X^2,(7,10)], [X^2,(7,11)], [X^2,(7,12)]) \\
sage: & factor(charpoly(M.T(2))) \\
(x - 7) * (x + 7) * (x - 9)^2 * (x + 5)^2 * (x^2 - x - 4)^2 * (x^2 + 9)^2 \\
sage: & dimension(M.cuspidal_subspace()) \\
10
\end{align*}
\]

{Cremona’s Modular Symbols Library} Sage includes John Cremona’s specialized and insanely fast implementation of modular symbols for weight 2 and trivial character. We illustrate below computing the space of modular symbols of

12.1. Thematic tutorial document tree
level 20014, which has dimension 5005, along with a Hecke operator on this space. The whole computation below takes only a few seconds; a similar computation takes a few minutes using Sage’s generic modular symbols code. Moreover, Cremona has done computations at levels over 200,000 using his library, so the code is known to scale well to large problems. The new code in Sage for modular symbols is much more general, but doesn’t scale nearly so well (yet).

\[\text{sage: } M = \text{CremonaModularSymbols}(20014) \quad \# \text{ few seconds}\]
\[\text{sage: } M\]
Cremona Modular Symbols space of dimension 5005 for Gamma_0(20014) of weight 2 with sign 0
\[\text{sage: } t = M.\text{hecke_matrix}(3) \quad \# \text{ few seconds}\]

### Enumerating Totally Real Number Fields

As part of his project to enumerate Shimura curves, John Voight has contributed code to Sage for enumerating totally real number fields. The algorithm isn’t extremely complicated, but it involves some “inner loops” that have to be coded to run very quickly. Using Cython, Voight was able to implement exactly the variant of Newton iteration that he needed for his problem.

The function \texttt{enumerate_totallyreal_fields_prim(n, B, \ldots)} enumerates without using a database (!) primitive (no proper subfield) totally real fields of degree \(n \geq 1\) with discriminant \(d \leq B\).

We compute the totally real quadratic fields of discriminant \(\leq 50\). The calculation below, which is almost instant, is done in real time and is not a table lookup.

\[\text{sage: } \text{enumerate_totallyreal_fields_prim}(2, 50)\]
\[[[5, x^2 - x - 1], [8, x^2 - 2], [12, x^2 - 3], [13, x^2 - x - 3],
 [17, x^2 - x - 4], [21, x^2 - x - 5], [24, x^2 - 6], [28, x^2 - 7],
 [29, x^2 - x - 7], [33, x^2 - x - 8], [37, x^2 - x - 9],
 [40, x^2 - 10], [41, x^2 - x - 10], [44, x^2 - 11]]\]

We compute all totally real quintic fields of discriminant \(\leq 10^5\). Again, this is done in real time - it’s not a table lookup!

\[\text{sage: } \text{enumerate_totallyreal_fields_prim}(5, 10^5)\]
\[[[14641, x^5 - x^4 - 4*x^3 + 3*x^2 + 3*x - 1],
 [24217, x^5 - 5*x^3 - x^2 + 3*x + 1],
 [36497, x^5 - 2*x^4 - 3*x^3 + 5*x^2 + x - 1],
 [38569, x^5 - 5*x^3 + 4*x - 1],
 [65657, x^5 - x^4 - 5*x^3 + 2*x^2 + 5*x + 1],
 [70601, x^5 - x^4 - 5*x^3 + 2*x^2 + 3*x - 1],
 [81509, x^5 - x^4 - 5*x^3 + 3*x^2 + 5*x - 2],
 [81589, x^5 - 6*x^3 + 8*x - 1],
 [89417, x^5 - 6*x^3 - x^2 + 8*x + 3]]\]

### Bernoulli Numbers

#### Mathematica and Pari

From the Mathematica website:

“Today We Broke the Bernoulli Record: From the Analytical Engine to Mathematica April 29, 2008 Oleksandr Pavlyk, Kernel Technology A week ago, I took our latest development version of Mathematica, and I typed \texttt{BernoulliB[10^7]}. And then I waited. Yesterday–5 days, 23 hours, 51 minutes, and 37 seconds later–I got the result!”
Tom Boothby did that same computation in Sage, which uses Pari’s bernfrac command that uses evaluation of \( \zeta \) and factorial to high precision, and it took 2 days, 12 hours.

**David Harvey’s bernmm**

Then David Harvey came up with an entirely new algorithm that parallelizes well. He gives these timings for computing \( B_{10^7} \) on his machine (it takes 59 minutes, 57 seconds on my 16-core 1.8ghz Opteron box):

PARI: 75 h, Mathematica: 142 h

bernm (1 core) = 11.1 h, bernmm (10 cores) = 1.3 h

“Running on 10 cores for 5.5 days, I [David Harvey] computed \( B_k \) for \( k = 10^8 \), which I believe is a new record. Essentially it’s the multimodular algorithm I suggested earlier on this thread, but I figured out some tricks to optimise the crap out of the computation of \( B_k \mod p \).”

So now Sage is the fastest in the world for large Bernoulli numbers. The timings below are on a 24-core 2.66Ghz Xeon box.

```
sage: w1 = bernoulli(100000, num_threads=16)  # 0.9 seconds wall time
sage: w2 = bernoulli(100000, algorithm='pari')  # long time (6s on sage.math, 2011)
sage: w1 == w2  # long time
True
```

### Polynomial Arithmetic

**FLINT: Univariate Polynomial Arithmetic**

Sage uses Bill Hart and David Harvey’s GPL’d Flint C library for arithmetic in \( \mathbb{Z}[x] \). Its main claim to fame is that it is the world’s fastest for polynomial multiplication, e.g., in the benchmark below it is faster than NTL and Magma on some systems (though such benchmarks of course change as software improves). Behind the scenes Flint contains some carefully tuned discrete Fourier transform code.

```
sage: Rflint = PolynomialRing(ZZ, 'x')
sage: f = Rflint([ZZ.random_element(2^64) for _ in [1..32]])
sage: g = Rflint([ZZ.random_element(2^64) for _ in [1..32]])
sage: timeit('f*g')  # random output
625 loops, best of 3: 105 microseconds per loop
sage: Rntl = PolynomialRing(ZZ, 'x', implementation='NTL')
sage: f = Rntl([ZZ.random_element(2^64) for _ in [1..32]])
sage: g = Rntl([ZZ.random_element(2^64) for _ in [1..32]])
sage: timeit('f*g')  # random output
625 loops, best of 3: 310 microseconds per loop
sage: ff = magma(f); gg = magma(g)  # optional - magma
sage: s = 'time v := [%s * %s : i in [1..10^5]];%(ff.name(), gg.name())' % (ff.name(), gg.name())  # optional - magma
sage: magma.eval(s)  # optional - magma
'Time: ...'
```
Singular: Multivariate Polynomial Arithmetic

Multivariate polynomial arithmetic in many cases uses Singular in library mode (due to Martin Albrecht), which is quite fast. For example, below we do the Fateman benchmark over the finite field of order 32003, and compare the timing with Magma.

```
sage: P.<x,y,z> = GF(32003)[]
sage: p = (x+y+z+1)^10
sage: q = p+1
sage: timeit('p*q')  # random output
125 loops, best of 3: 1.53 ms per loop
sage: p = (x+y+z+1)^20
sage: q = p+1
sage: timeit('p*q')  # not tested - timeout if SAGE_DEBUG=yes
5 loops, best of 3: 384 ms per loop
sage: pp = magma(p); qq = magma(q)  #optional - magma
sage: s = 'time w := %s*%s;'%(pp.name(),qq.name())  #optional - magma
sage: magma.eval(s)  #optional - magma
'Time: ...
```

Modular Forms

This section is about computing with modular forms, modular symbols, and modular abelian varieties. Most of the Sage functionality we describe below is new code written for Sage by myself, Craig Citro, Robert Bradshaw, and Jordi Quer in consultation with John Cremona. It has much overlap in functionality with the modular forms code in Magma, which I developed during 1998-2004.

Modular Forms and Hecke Operators

Congruence subgroups

Definition

A congruence subgroup is a subgroup of the group $\text{SL}_2(\mathbb{Z})$ of determinant $\pm 1$ integer matrices that contains

$$\Gamma(N) = \text{Ker}(\text{SL}_2(\mathbb{Z}) \to \text{SL}_2(\mathbb{Z}/N\mathbb{Z}))$$

for some positive integer $N$. Since $\Gamma(N)$ has finite index in $\text{SL}_2(\mathbb{Z})$, all congruence subgroups have finite index. The converse is not true, though in many other settings it is true (see [paper of Serre]).

The inverse image $\Gamma_0(N)$ of the subgroup of upper triangular matrices in $\text{SL}_2(\mathbb{Z}/N\mathbb{Z})$ is a congruence subgroup, as is the inverse image $\Gamma_1(N)$ of the subgroup of matrices of the form $\left(\begin{smallmatrix} 1 & * \\ 0 & 1 \end{smallmatrix}\right)$. Also, for any subgroup $H \subset (\mathbb{Z}/N\mathbb{Z})^\times$, the inverse image $\Gamma_H(N)$ of the subgroup of $\text{SL}_2(\mathbb{Z}/N\mathbb{Z})$ of all elements of the form $\left(\begin{smallmatrix} a & * \\ 0 & d \end{smallmatrix}\right)$ with $d \in H$ is a congruence subgroup.

We can create each of the above congruence subgroups in Sage, using the Gamma0, Gamma1, and GammaH commands.
The second argument to the GammaH command is a list of generators of the subgroup $H$ of $(\mathbb{Z}/N\mathbb{Z})^*$.

Generators

Sage can compute a list of generators for these subgroups. The algorithm Sage uses is a straightforward generic procedure that uses coset representatives for the congruence subgroup (which are easy to enumerate) to obtain a list of generators [[ref my modular forms book]].

The list of generators Sage computes is unfortunately large. Improving this would be an excellent Sage development project, which would involve much beautiful mathematics.

UPDATE (March 2012): The project referred to above has been carried out (by several people, notably Hartmut Monien, building on earlier work of Chris Kurth). Sage now uses a much more advanced algorithm based on Farey symbols which calculates a minimal set of generators.

Modular Forms

Definition

A modular form on a congruence subgroup $\Gamma$ of integer weight $k$ is a holomorphic function $f(z)$ on the upper half plane

$$\mathfrak{h}^* = \{z \in \mathbb{C} : \Im(z) > 0\} \cup \mathbb{Q} \cup \{i\infty\}$$

such that for every matrix $\begin{pmatrix} a & b \\ c & d \end{pmatrix} \in \Gamma$, we have

$$f \left( \frac{az + b}{cz + d} \right) = (cz + d)^k f(z).$$

A cusp form is a modular form that vanishes at all of the cusps $\mathbb{Q} \cup \{i\infty\}$. 

12.1. Thematic tutorial document tree
If $\Gamma$ contains $\Gamma_1(N)$ for some $N$, then $\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \in \Gamma$, so the modular form condition implies that $f(z) = f(z + 1)$. This, coupled with the holomorphicity condition, implies that $f(z)$ has a Fourier expansion

$$f(z) = \sum_{n=0}^{\infty} a_n e^{2\pi i nz}$$

with $a_n \in \mathbb{C}$. We let $q = e^{2\pi i z}$, and call $f = \sum_{n=0}^{\infty} a_n q^n$ the $q$-expansion of $f$.

**Creation in Sage**

Henceforth we assume that $\Gamma$ is either $\Gamma_1(N)$, $\Gamma_0(N)$, or $\Gamma_H(N)$ for some $H$ and $N$. The complex vector space $M_k(\Gamma)$ of all modular forms of weight $k$ on $\Gamma$ is a finite dimensional vector space.

We create the space $M_k(\Gamma)$ in Sage by typing `ModularForms(G, k)` where $G$ is the congruence subgroup and $k$ is the weight.

```sage
sage: ModularForms(Gamma0(25), 4)
Modular Forms space of dimension 11 for Gamma(25) of weight 4 and character [0, 0] (mod 25)
sage: S = CuspForms(Gamma0(25),4, prec=15); S
Cuspidal subspace of dimension 5 of Modular Forms space of dimension 11 for Gamma(25) of weight 4 and character [0, 0] (mod 25)
sage: S.basis()
[q + q^9 - 8*q^11 - 8*q^14 + O(q^15),
 q^2 - q^7 - 7*q^12 + 7*q^13 + O(q^15),
 q^3 + q^7 - 2*q^8 - 6*q^12 - 5*q^13 + O(q^15),
 q^4 - q^6 - 3*q^9 + 5*q^11 - 2*q^14 + O(q^15),
 q^5 - 4*q^10 + O(q^15)]
```

**Dimension Formulas**

Sage computes the dimensions of all these spaces using simple arithmetic formulas instead of actually computing bases for the spaces in question. In fact, Sage has the most general collection of modular forms dimension formulas of any software; type `help(sage.modular.dims)` to see a list of arithmetic functions that are used to implement these dimension formulas.

```sage
sage: ModularForms(Gamma1(949284), 456).dimension()
11156973844800
sage: from sage.modular.dims import dimension_cusp_forms
sage: a = [dimension_cusp_forms(Gamma0(N),2) for N in [1..25]]; a
[0, 0, ..., 1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 2, 2, 1, 0]
sage: oeis(a)  # optional - internet
0: A001617: Genus of modular group Gamma_0(N). Or, genus of modular curve X_0(N).
```

Sage does not have simple formulas for dimensions of spaces of modular forms of weight 1, since such formulas perhaps do not exist.
Diamond Bracket Operators

The space $M_k(\Gamma_1(N))$ is equipped with an action of $(\mathbb{Z}/N\mathbb{Z})^*$ by diamond bracket operators $\langle d \rangle$, and this induces a decomposition

$$M_k(\Gamma_1(N)) = \bigoplus_{\varepsilon:(\mathbb{Z}/N\mathbb{Z})^* \to \mathbb{C}^*} M_k(N, \varepsilon),$$

where the sum is over all complex characters of the finite abelian group $(\mathbb{Z}/N\mathbb{Z})^*$. These characters are called Dirichlet characters, which are central in number theory.

The factors $M_k(N, \varepsilon)$ then have bases whose $q$-expansions are elements of $R[[q]]$, where $R = \mathbb{Z}[\varepsilon]$ is the ring generated over $\mathbb{Z}$ by the image of $\varepsilon$. We illustrate this with $N = k = 5$ below, where DirichletGroup will be described later.

```sage
sage: CuspForms(DirichletGroup(5).0, 5).basis()
[q + (-zeta4 - 1)*q^2 + (6*zeta4 - 6)*q^3 - ... + O(q^6)]
```

Dirichlet Characters

Use the command `DirichletGroup(N,R)` to create the group of all Dirichlet characters of modulus $N$ taking values in the ring $R$. If $R$ is omitted, it defaults to a cyclotomic field.

```sage
sage: G = DirichletGroup(8); G
Group of Dirichlet characters modulo 8 with values in Cyclotomic Field of order 2 and degree 1
sage: v = G.list(); v
[Dirichlet character modulo 8 of conductor 1 mapping 7 |--> 1, 5 |--> 1,
 Dirichlet character modulo 8 of conductor 4 mapping 7 |--> -1, 5 |--> 1,
 Dirichlet character modulo 8 of conductor 8 mapping 7 |--> 1, 5 |--> -1,
 Dirichlet character modulo 8 of conductor 8 mapping 7 |--> -1, 5 |--> -1]
sage: eps = G.0; eps
Dirichlet character modulo 8 of conductor 4 mapping 7 |--> -1, 5 |--> 1
sage: eps.values()
[0, 1, 0, -1, 0, 1, 0, -1]
```

Sage both represents Dirichlet characters by giving a “matrix”, i.e., the list of images of canonical generators of $(\mathbb{Z}/N\mathbb{Z})^*$, and as vectors modulo an integer $n$. For years, I was torn between these two representations, until J. Quer and I realized that the best approach is to use both and make it easy to convert between them.

```sage
sage: parent(eps.element())
Vector space of dimension 2 over Ring of integers modulo 2
```

Given a Dirichlet character, Sage also lets you compute the associated Jacobi and Gauss sums, generalized Bernoulli numbers, the conductor, Galois orbit, etc.
Decomposing $M_k(\Gamma_1(N))$

Recall that Dirichlet characters give a decomposition

$$M_k(\Gamma_1(N)) = \bigoplus_{\epsilon: (\mathbb{Z}/N\mathbb{Z})^* \rightarrow \mathbb{C}^*} M_k(N, \epsilon).$$

Given a Dirichlet character $\epsilon$ we type `ModularForms(eps, weight)` to create the space of modular forms with that character and a given integer weight. For example, we create the space of forms of weight 5 with the character modulo 8 above that is $-1$ on 3 and 1 on 5 as follows.

```
sage: ModularForms(eps,5)
Modular Forms space of dimension 6, character [-1, 1] and
weight 5 over Rational Field
sage: sum([ModularForms(eps,5).dimension() for eps in v])
11
sage: ModularForms(Gamma1(8),5)
Modular Forms space of dimension 11 ...
```

**Note:** Exercise: Compute the dimensions of all spaces $M_2(37, \epsilon)$ for all Dirichlet characters $\epsilon$.

**Hecke Operators**

The space $M_k(\Gamma)$ is equipped with an action of a commuting ring $\mathbb{T}$ of Hecke operators $T_n$ for $n \geq 1$. A standard computational problem in the theory of modular forms is to compute an explicit basis of $q$-expansion for $M_k(\Gamma)$ along with matrices for the action of any Hecke operator $T_n$, and to compute the subspace $S_k(\Gamma)$ of cusp forms.

```
sage: M = ModularForms(Gamma0(11),4)
sage: M.basis()
[q + 3*q^3 - 6*q^4 - 7*q^5 + O(q^6),
q^2 - 4*q^3 + 2*q^4 + 8*q^5 + O(q^6),
1 + 0(q^6),
q + 9*q^2 + 28*q^3 + 73*q^4 + 126*q^5 + O(q^6)]
sage: M.hecke_matrix(2)
[0 2 0 0]
[1 2 0 0]
[0 0 9 0]
[0 0 0 9]
sage: M.hecke_matrix(3)
[ 3 -8]
[-4 -5]
```

We can also compute Hecke operators on the cuspidal subspace.

```
sage: S = M.cuspidal_subspace()
sage: S.hecke_matrix(2)
[0 2]
[1 2]
sage: S.hecke_matrix(3)
[ 3 -8]
[-4 -5]
```
Hecke Operator on $M_2(\Gamma_1(N))$

At the time these lectures were first written, Sage didn’t yet implement computation of the Hecke operators on $M_k(\Gamma_1(N))$, but these have subsequently been added:

```python
sage: M = ModularForms(Gamma1(5),2)
sage: M
Modular Forms space of dimension 3 for Congruence Subgroup Gamma1(5) of weight 2 over Rational Field
sage: M.hecke_matrix(2)
\begin{bmatrix}
-21 & 0 & -240 \\
-2 & 0 & -23 \\
2 & 1 & 24
\end{bmatrix}
```

These are calculated by first calculating Hecke operators on modular symbols for $\Gamma_1(N)$, which is a $\mathbb{T}$-module that is isomorphic to $M_k(\Gamma_1(N))$ (see Modular Symbols).

```python
sage: ModularSymbols(Gamma1(5),2,sign=1).hecke_matrix(2)
\begin{bmatrix}
2 & 1 & 1 \\
1 & 2 & -1 \\
0 & 0 & -1
\end{bmatrix}
```

Modular Symbols

Modular symbols are a beautiful piece of mathematics that was developed since the 1960s by Birch, Manin, Shokorov, Mazur, Merel, Cremona, and others. Not only are modular symbols a powerful computational tool as we will see, they have also been used to prove rationality results for special values of $L$-series, to construct $p$-adic $L$-series, and they play a key role in Merel’s proof of the uniform boundedness theorem for torsion points on elliptic curves over number fields.

We view modular symbols as a remarkably flexible computational tool that provides a single uniform algorithm for computing $M_k(N,\varepsilon)$ for any $N, \varepsilon$ and $k \geq 2$. There are ways to use computation of those spaces to obtain explicit basis for spaces of weight 1 and half-integral weight, so in a sense modular symbols yield everything. There are also generalizations of modular symbols to higher rank groups, though Sage currently has no code for modular symbols on higher rank groups.

Definition

A modular symbol of weight $k$, and level $N$, with character $\varepsilon$ is a sum of terms $X^i Y^{k-2-i}\{\alpha, \beta\}$, where $0 \leq i \leq k-2$ and $\alpha, \beta \in \mathbb{P}^1(Q) = Q \cup \{\infty\}$. Modular symbols satisfy the relations

$$X^i Y^{k-2-i}\{\alpha, \beta\} + X^i Y^{k-2-i}\{\beta, \gamma\} + X^i Y^{k-2-i}\{\gamma, \alpha\} = 0$$

and for every $\gamma = \left(\begin{array}{cc}a & b \\c & d\end{array}\right) \in \Gamma_0(N)$, we have

$$(dX - bY)^i(-cX + aY)^{k-2-i}\{\gamma(\alpha), \gamma(\beta)\} = \varepsilon(d) X^i Y^{k-2-i}\{\alpha, \beta\}.$$ 

The modular symbols space $\mathcal{M}_k(N,\varepsilon)$ is the torsion free $Q[\varepsilon]$-module generated by all sums of modular symbols, modulo the relations listed above. Here $Q[\varepsilon]$ is the ring generated by the values of the character $\varepsilon$, so it is of the form $Q[\zeta_m]$ for some integer $m$. 

12.1. Thematic tutorial document tree 505
The amazing theorem that makes modular symbols useful is that there is an explicit description of an action of a Hecke algebra $T$ on $M_k(N, \varepsilon)$, and there is an isomorphism

$$M_k(N, \varepsilon; \mathbb{C}) \cong M_k(N, \varepsilon) \oplus S_k(N, \varepsilon).$$

This means that if modular symbols are computable (they are!), then they can be used to compute a lot about the $T$-module $M_k(N, \varepsilon)$.

**Manin Symbols**

**Definition**

Though $M_k(N, \varepsilon)$ as described above is not explicitly generated by finitely many elements, it is finitely generated. Manin, Shokoruv, and Merel give an explicit description of finitely many generators (Manin symbols) for this space, along with all explicit relations that these generators satisfy (see my book). In particular, if we let

$$(i, c, d) = [X^iY^{2-k-i}, (c, d)] = (dX - bY)^i(-cX + aY)^{k-2-i}\{\gamma(0), \gamma(\infty)\},$$

where $\gamma = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$, then the Manin symbols $(i, c, d)$ with $0 \leq i \leq k-2$ and $(c, d) \in \mathbb{P}^1(N)$ generate $M_k(N, \varepsilon)$.

**Computing in Sage**

We compute a basis for the space of weight 4 modular symbols for $\Gamma_0(11)$, then coerce in $(2,0,1)$ and $(1,1,3)$.

```sage
sage: M = ModularSymbols(11, 4)
sage: M.basis()
[(X^2,(0,1)], [X^2,(1,6)], [X^2,(1,7)], [X^2,(1,8)], [X^2,(1,9)], [X^2,(1,10)]]

sage: M((2,0,1))
[X^2,(0,1)]

sage: M((1,1,3))
2/7*[X^2,(1,6)] + 1/14*[X^2,(1,7)] - 4/7*[X^2,(1,8)] + 3/14*[X^2,(1,10)]
```

We compute a modular symbols representation for the Manin symbol $(2, 1, 6)$, and verify this by converting back.

```sage
sage: a = M.1; a
[X^2,(1,6)]

sage: a.modular_symbol_rep()
36*X^2*{-1/6, 0} + 12*X^2*Y^2*{-1/6, 0} + Y^2*{-1/6, 0}

sage: 36*M([2,-1/6,0]) + 12*M([1,-1/6,0]) + M([0,-1/6,0])
[X^2,(1,6)]
```

**Method of Graphs**

The Mestre Method of Graphs is an intriguing algorithm for computing the action of Hecke operators on yet another module $X$ that is isomorphic to $M_2(\Gamma_0(N))$. The implementation in Sage unfortunately only works when $N$ is prime; in contrast, my implementation in Magma works when $N = pM$ and $S_2(\Gamma_0(M)) = 0$.

The matrices of Hecke operators on $X$ are vastly sparser than on any basis of $M_2(\Gamma_0(N))$ that you are likely to use.
sage: X = SupersingularModule(389); X
Module of supersingular points on X_0(1)/F_389 over Integer Ring
sage: t2_mestre = X.T(2).matrix()
sage: p_mestre = t2_mestre.charpoly()
sage: factor(p_mestre)
(x - 3) * (x + 2) * (x^2 - 2) * (x^3 - 4*x - 2) * ...

sage: t2 = ModularSymbols(389,sign=1).hecke_matrix(2)
sage: p = t2.charpoly()
sage: factor(p)
(x - 3) * (x + 2) * (x^2 - 2) * (x^3 - 4*x - 2) * ...

sage: p_mestre == p
True

The method of graphs is also used in computer science to construct expander graphs with good properties. And it is important in my algorithm for computing Tamagawa numbers of purely toric modular abelian varieties. This algorithm is not implemented in Sage yet, since it is only interesting in the case of non-prime level, as it turns out.

REFERENCE:
• Jean-François Mestre, La méthode des graphes. Exemples et applications, Proceedings of the international conference on class numbers and fundamental units of algebraic number fields (Katata, 1986), 217-242

Level One Modular Forms

Computing $\Delta$

The modular form

$$\Delta = q \prod (1 - q^n)^{24} = \sum \tau(n)q^n$$

is perhaps the world’s most famous modular form. We compute some terms from the definition.

sage: R.<q> = QQ[[[]]

sage: q * prod(1-q^n+O(q^6) for n in (1..5))^24
q - 24*q^2 + 252*q^3 - 1472*q^4 + 4830*q^5 - 6048*q^6 + O(q^7)

There are much better ways to compute $\Delta$, which amount to just a few polynomial multiplications over $\mathbb{Z}$.

sage: D = delta_qexp(10^5) # less than 10 seconds
sage: D[:10]
q - 24*q^2 + 252*q^3 - 1472*q^4 + ...

sage: [p for p in primes(10^5) if D[p] % p == 0]
[2, 3, 5, 7, 2411]

sage: D[2411]
4542041100095889012

sage: f = eisenstein_series_qexp(12,6) - D[:6]; f
691/65520 + 2073*q^2 + 176896*q^3 + 4197825*q^4 + 48823296*q^5 + O(q^6)

sage: f % 691
0(q^6)
The Victor Miller Basis

The Victor Miller basis for \( M_k(\text{SL}_2(\mathbb{Z})) \) is the reduced row echelon basis. It’s a lemma that it has all integer coefficients, and a rather nice diagonal shape.

```
sage: victor_miller_basis(24, 6)
[1 + 52416000*q^3 + 39007332000*q^4 + 6609020221440*q^5 + O(q^6),
  q + 195660*q^3 + 12080128*q^4 + 44656110*q^5 + O(q^6),
  q^2 - 48*q^3 + 1080*q^4 - 15040*q^5 + O(q^6)]
```

```
sage: dimension_modular_forms(1, 200)
17
```

```
sage: B = victor_miller_basis(200, 18)  #5 seconds
sage: B
[1 + 792883144206817340486660707200000*q^17 + O(q^18),
  q + 268760271810677283792896846869*q^17 + O(q^18),
  ...
  q^16 + 96*q^17 + O(q^18)]
```

Note: Craig Citro has made the above computation an order of magnitude faster in code he has not quite got into Sage yet.

“I’ll clean those up and submit them soon, since I need them for something I’m working on … I’m currently in the process of making spaces of modular forms of level one subclass the existing code, and actually take advantage of all our fast \( E_k \) and \( \Delta \) computation code, as well as cleaning things up a bit.”

Half Integral Weight Forms

Basmaji’s Algorithm


Let \( S = S_{k+1}(\varepsilon) \) be the space of cusp forms of even integer weight \( k + 1 \) and character \( \varepsilon = \chi\psi^{(k+1)/2} \), where \( \psi \) is the nontrivial mod-4 Dirichlet character. Let \( U \) be the subspace of \( S \times S \) of elements \((a, b)\) such that \( \Theta_2a = \Theta_3b \). Then \( U \) is isomorphic to \( S_{k/2}(\chi) \) via the map \((a, b) \mapsto a/\Theta_3\).

This algorithm is implemented in Sage. I’m sure it could be implemented in a way that is much faster than the current implementation...

```
sage: half_integral_weight_modform_basis(DirichletGroup(16, QQ).1, 3, 10)
[]
sage: half_integral_weight_modform_basis(DirichletGroup(16, QQ).1, 5, 10)
[q - 2*q^3 - 2*q^5 + 4*q^7 - q^9 + O(q^10)]
sage: half_integral_weight_modform_basis(DirichletGroup(16*7).0^2, 3, 30)
[q^2 - q^2 - q^9 + 2*q^14 + 6*q^18 - 2*q^21 - 4*q^22 - q^25 + O(q^30),
  q^2 - q^14 - 3*q^18 + 2*q^22 + O(q^30),
  q^4 - q^8 - q^16 + q^28 + O(q^30), q^7 - 2*q^15 + O(q^30)]
```
Generators for Rings of Modular Forms

Computing Generators

For any congruence subgroup \( \Gamma \), the direct sum

\[ M(\Gamma) = \bigoplus_{k \geq 0} M_k(\Gamma) \]

is a ring, since the product of modular forms \( f \in M_k(\Gamma) \) and \( g \in M_{k'}(\Gamma) \) is an element \( fg \in M_{k+k'}(\Gamma) \). Sage can compute likely generators for rings of modular forms, but currently doesn’t prove any of these results.

We verify the statement proved in Serre’s “A Course in Arithmetic” that \( E_4 \) and \( E_6 \) generate the space of level one modular forms.

```sage
ModularFormsRing(SL2Z).generators(prec=4)
[(4, 1 + 240*q + 2160*q^2 + 6720*q^3 + O(q^4)),
(6, 1 - 504*q - 16632*q^2 - 122976*q^3 + O(q^4))]
```

Have you ever wondered which forms generate the ring \( M(\Gamma_0(2)) \)? It turns out that one form of weight 2 and one form of weight 4 suffice.

```sage
ModularFormsRing(Gamma0(2)).generators(prec=12)
[(2, 1 + 12*q + 36*q^2 + 12*q^3 + 84*q^4 + 72*q^5 + 36*q^6 + 96*q^7 + 180*q^8 + 12*q^9 + O(q^10)),
(4, 1 + 240*q^3 + 2160*q^6 + 6720*q^9 + O(q^10))]
```

Here’s generators for \( M(\Gamma_0(3)) \). Notice that elements of weight 6 are now required, in addition to weights 2 and 4.

```sage
ModularFormsRing(Gamma0(3)).generators()
[(2, 1 + 12*q + 36*q^2 + 12*q^3 + 84*q^4 + 72*q^5 + 36*q^6 + 96*q^7 + 180*q^8 + 12*q^9 + O(q^10)),
(3, 1 + 54*q^2 + 72*q^3 + 432*q^5 + 270*q^6 + 918*q^8 + 720*q^9 + O(q^10)),
(6, 1 - 504*q^3 - 16632*q^6 - 122976*q^9 + O(q^10))]
```

(Note: As of 2012, updates to the code mean that the output of this test is not quite the same as it was in 2008, but of course there are multiple equally valid answers.)

We can also handle rings of modular forms for odd congruence subgroups, but with the usual caveat that we can’t calculate forms of weight 1. So these are elements generating the graded ring of forms of weight 0 or \( \geq 2 \).

```sage
ModularFormsRing(Gamma1(3)).generators()
[(2, 1 + 12*q + 36*q^2 + 12*q^3 + 84*q^4 + 72*q^5 + 36*q^6 + 96*q^7 + 180*q^8 + 12*q^9 + O(q^10)),
(3, 1 + 54*q^2 + 72*q^3 + 432*q^5 + 270*q^6 + 918*q^8 + 720*q^9 + O(q^10)),
(3, q + 9*q^2 + 13*q^4 + 24*q^5 + 27*q^6 + 50*q^7 + 51*q^8 + 81*q^9 + O(q^10)),
(4, 1 + 240*q^3 + 2160*q^6 + 6720*q^9 + O(q^10))]
```
Thematic Tutorials, Release 9.7

\*L*-series

\*L*-series of \(\Delta\)

Thanks to wrapping work of Jennifer Balakrishnan of M.I.T., we can compute explicitly with the \(L\)-series of the modular form \(\Delta\). Like for elliptic curves, behind these scenes this uses Dokchitsers \(L\)-functions calculation Pari program.

```python
sage: L = delta_lseries(); L
L-series of conductor 1 and weight 12
sage: L(1)
0.0374412812685155
```

\*L*-series of a Cusp Form

In some cases we can also compute with \(L\)-series attached to a cusp form.

```python
sage: f = CuspForms(2,8).newforms()[0]
sage: L = f.lseries()
sage: L(1)
0.0884317737041015
sage: L(0.5)
0.0296568512531983
```

\*L*-series of a General Newform is Not Implemented

Unfortunately, computing with the \(L\)-series of a general newform is not yet implemented.

```python
sage: S = CuspForms(23,2); S
Cuspidal subspace of dimension 2 of Modular Forms space of dimension 3 for Congruence Subgroup Gamma0(23) of weight 2 over Rational Field
sage: f = S.newforms('a')[0]; f
q + a0*q^2 + (-2*a0 - 1)*q^3 + (-a0 - 1)*q^4 + 2*a0*q^5 + O(q^6)
```

Computing with \(L(f, s)\) totally not implemented yet, though should be easy via Dokchitser.

\section*{Modular Abelian Varieties}

The quotient of the extended upper half plane \(\mathbb{H}^*\) by the congruence subgroup \(\Gamma_1(N)\) is the modular curve \(X_1(N)\). Its Jacobian \(J_1(N)\) is an abelian variety that is canonically defined over \(\mathbb{Q}\). Likewise, one defines a modular abelian variety \(J_0(N)\) associated to \(\Gamma_0(N)\).

A modular abelian variety is an abelian variety over \(\mathbb{Q}\) that is a quotient of \(J_1(N)\) for some \(N\).

The biggest recent theorem in number theory is the proof of Serre’s conjecture by Khare and Wintenberger. According to an argument of Ribet and Serre, this implies the following modularity theorem, which generalizes the modularity theorem that Taylor-Wiles proved in the course of proving Fermat’s Last Theorem.

One of my long-term research goals is to develop a systematic theory for computing with modular abelian varieties. A good start is the observation using the Abel-Jacobi theorem that every modular abelian variety (up to isomorphism) can be specified by giving a lattice in a space of modular symbols.
Computing in Sage

We define some modular abelian varieties of level 39, and compute some basic invariants.

```
sage: D = J0(39).decomposition(); D
[Simple abelian subvariety 39a(1,39) of dimension 1 of J0(39),
  Simple abelian subvariety 39b(1,39) of dimension 2 of J0(39)]
sage: D[1].lattice()
Free module of degree 6 and rank 4 over Integer Ring
Echelon basis matrix:
[ 1 0 0 1 -1 0]
[ 0 1 1 0 -1 0]
[ 0 0 2 0 -1 0]
[ 0 0 0 0 0 1]
sage: G = D[1].rational_torsion_subgroup(); G
Torsion subgroup of Simple abelian subvariety 39b(1,39)
of dimension 2 of J0(39)
sage: G.order()
28
sage: G.gens()
[[1/14, 2/7, 0, 1/14, -3/14, 1/7],
  [0, 1, 0, 0, -1/2, 0],
  [0, 0, 1, 0, -1/2, 0]]
sage: B, phi = D[1]/G
sage: B
Abelian variety factor of dimension 2 of J0(39)
sage: phi.kernel()
(Finite subgroup with invariants [2, 14] ...
```

Endomorphisms

There is an algorithm in Sage for computing the exact endomorphism ring of any modular abelian variety.

```
sage: A = J0(91)[2]; A
Simple abelian subvariety 91c(1,91) of dimension 2 of J0(91)
sage: R = End(A); R
Endomorphism ring of Simple abelian subvariety 91c(1,91)
of dimension 2 of J0(91)
sage: for x in R.gens():
    ....:    print(x.matrix())
    ....:    print("")
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
[0 4 -2 0]
[-1 5 -2 1]
[-1 2 0 2]
[-1 1 0 3]
```
It is also possible to test isomorphism of two modular abelian varieties. But much exciting theoretical and computational work remains to be done.

## 12.1.21 Profiling in Sage

This page lists several methods available in Sage to measure and analyze the performances of a piece of code. For more general information on profiling, see [Wikipedia article Profiling](https://en.wikipedia.org/wiki/Profiling_(computer_programming)).

### Table of contents

- **Profiling in Sage**
  - How long does it take? %time and %timeit
  - Python-level function calls: %prun
  - Python-level line-by-line profiling: %lprun
  - C-level function calls: %crun
  - C-level line-by-line profiling: perf (Linux only)

### How long does it take? %time and %timeit

The two IPython magics %time and %timeit measure the time it takes to run a command:

```
  sage: %time p=random_prime(2**300)
  CPU times: user 152 ms, sys: 0 ns, total: 152 ms
  Wall time: 150 ms

  sage: %timeit p=random_prime(2**300)
  10 loops, best of 3: 62.2 ms per loop
```

Note that while %time only runs the command once, %timeit tries to return a more meaningful value over several runs.

For more information see %timeit? or [this page](https://docs.python.org/3/library/timeit.html).

Note that Sage provides a `timeit` function which also runs in the Sage notebook.

### Python-level function calls: %prun

With %prun, you can obtain the list of all Python functions involved in a computation, as well as the time spent on each of them:

```
  sage: %prun _=random_prime(2**500)
  468 function calls in 0.439 seconds

  Ordered by: internal time

  ncalls  tottime  percall  cumtime  percall filename:lineno(function)
  32    0.438  0.014    0.438  0.014 {method 'is_prime' of 'sage.rings.integer._Integer' objects}
```
The most time-consuming functions should appear on the top. A description of the different columns is available here.

**Note:** You may want to sort this list differently, e.g: use `%prun -s cumulative` for decreasing cumulative time.

Alternatively, you can “save” this data to a `Stats` object for further inspection:

```
sage: %prun -r random_prime(2**500)
sage: stats_object = _
sage: stats_object.total_calls
```

For more information see `%prun?` or this page.

**Visualize the statistics:** you can obtain a more graphical output with RunSnake and Sage’s function `runsnake()`:

```
sage: runsnake('random_prime(2**500)')
```

**Python-level line-by-line profiling:** `%lprun`

With `line_profiler` and its `%lprun` magic, you can find out which lines of one (or many) functions are the most time-consuming. The syntax is the following:

```
%lprun -f function1 -f function2 code_to_run
```

This will display the line-by-line analysis of `function1` and `function2` when `code_to_run` is executed:

```
sage: %lprun -f random_prime random_prime(2**500)
```

In order to install `line_profiler` you must first run the following command:

```
[user@localhost ~] sage -pip install "line_profiler"
```
**C-level function calls: %crun**

With %crun, you can obtain the list of all C functions involved in a computation, as well as the time spent on each of them. You will need to have the Google performance analysis tools installed on your system:

```
sage: %crun p=random_prime(2**500)
PROFILE: interrupts/evictions/bytes = 45/0/18344
Total: 45 samples
  0  0.0%  0.0% 35  77.8% PyEval_EvalCode
  0  0.0%  0.0% 35  77.8% PyEval_EvalCodeEx
  0  0.0%  0.0% 35  77.8% PyEval_EvalFrameEx
  0  0.0%  0.0% 35  77.8% PyObject_Call
  0  0.0%  0.0% 35  77.8% PyRun_StringFlags
  0  0.0%  0.0% 35  77.8% __Pyx_PyObject_Call.constprop.73
...
```

For more information on %crun, see `sage.misc.gperftools`.

**C-level line-by-line profiling: perf (Linux only)**

If your code is written in C or in Cython, you can find out line-by-line which are the most costly using `perf` (included in the Ubuntu package `linux-tools`).

The easiest way to use it is to run some (very long) computation in Sage, and to type in a console

```
[user@localhost ~] sudo perf top
```

Select the entry that interests you, and press Enter. The annotate command will show you:

- the CPU instructions
- the source code
- the associated time

```
| * cdef unsigned long word = (<unsigned long>1) << (v & self.radix_mod_mask) |
| * return (self.edges[place] & word) >> (v & self.radix_mod_mask) |
| # <<<<<<<<<<<<<<<<
| *
| * cpdef bint has_arc(self, int u, int v) except -1: |
| */ |
| __pyx_r = (((__pyx_v_self->edges[__pyx_v_place]) & __pyx_v_word) >> (__pyx_v_v & __pyx_v_self->radix_mod_mask)); |
10.88 movslq %esi,%rsi
6.52 and (%rdi,%rsi,8),%rax
12.84 shr %cl,%rax
```

Note:

- press s to toggle source code view
- press H to cycle through hottest instructions
- press h for help
Alternatively, or if you have no sudo privileges, you can record the statistics of a specific process into a file \texttt{perf.data} from its PID. Then, visualize the result using \texttt{perf report}:

```
[user@localhost ~] perf record -p PID
[user@localhost ~] perf report --vmlinux vmlinux
```

### 12.1.22 Creating a Tutorial from an old Sage Worksheet (.sws)

A lot of pedagogical material has been written in the Sage Worksheet format, which is no longer supported by Sage after the transition to Python 3 and the removal of the SageNB package.

However, it is possible to convert Sage Worksheet files. Once you have created a worksheet and are satisfied with the text and computations, download it to a directory.

We will assume here that the worksheet is called \texttt{Tutorial.sws} and the directory is called \texttt{make_tutorial}. We also assume that \texttt{sage} is your Sage command; if it is not in your \texttt{PATH} then replace this with the path to your Sage installation, such as \texttt{/Applications/Sage-9.2.app/Contents/Resources/sage/sage} if you are using the Mac app and have placed it in your Applications directory.

- Next, you will need an optional package to convert your worksheet. Use the command:

\[
\texttt{sage -i sage_sws2rst}
\]

  to install it (or, in the Mac app, use the Terminal Session advanced menu).

- Then we will use the \texttt{sws2rst} script to turn the worksheet into a document in the ReStructuredText format. Be sure you are in the same directory as the worksheet:

\[
\texttt{sage --sws2rst Tutorial.sws}
\]

This will create an \texttt{.rst} file along with a subdirectory of image files (which may be empty if there are no images). You can find help for \texttt{sws2rst} with the command \texttt{sage --sws2rst -h} once you have installed beautifulsoup4.

- In principle, such a file could be added directly to Sage’s documentation (see the developer’s manual). However, you probably want to check whether it looks right first. So next we will compile this file to html documentation.

  - Follow the instructions of \texttt{sage --sws2rst --sphinxify}. First, we will open a Sage shell session, where all appropriate Sage references already work properly:

\[
\texttt{sage --sh}
\]

  From here, you should be able to just type:

\[
\texttt{sphinx-quickstart}
\]

  and then respond to prompts for turning your \texttt{.rst} file into documentation. For most of them you can just hit enter/return to accept the defaults. However, you will probably want to

  * Enter a name for the project
  * Enter a name for you
  * Type y for the question about using MathJax

Keep note of the instructions; the main other thing to do is add your file’s name to \texttt{index.rst}, and then just do:
and wait while magic happens. To see the results, open the file `make_tutorial/_build/html/Tutorial.html` with a browser, or use your graphical file system to navigate to the same place.

- Now you can modify the `.rst` file more and repeat the steps of compiling it until it is ready for inclusion, or just for distribution among other Sage users as an HTML file. (Do `make pdf` for a PDF version.)


[Levine2014] Lionel Levine. Threshold state and a conjecture of Poghosyan, Poghosyan, Priezzhev and Ruelle, Communications in Mathematical Physics. arXiv 1402.3283


