91 Sage

Robert A. Beezer
University of Puget Sound

Robert Bradshaw
Google

Jason Grout
Drake University

William Stein
University of Washington

An updated and expanded version of this chapter is available in the Sage documentation as a thematic tutorial on linear algebra.

91.1 Introduction

Sage is generally recognized as the leading comprehensive open source mathematics software system. Development began in 2005 by William Stein and progressed rapidly due to the incorporation of many mature open source packages for various areas of mathematics. The main web site for Sage is `http://sagemath.org`. Free downloads of Sage, free public servers, and many other resources are located at the web site.

Sage is distributed with an open-source license (GPL version 3), which provides several advantages over commercial offerings.

- There is no cost to obtain or upgrade Sage. Collaborators and students have a platform that does not depend on institutional licenses and remains accessible in the future.
- *All* source code can be freely examined, modified, and redistributed. Algorithms can be traced and examined, bugs can be located, and improvements and bugfixes can be made and distributed freely. Often, bugs and improvements are made by experts in the field and frequent updates make these available rapidly.
- Every copy includes the web server software to run Sage as a multi-user server, so workgroups and educational institutions can provide computational services over the Internet at no additional cost.
- Many routines in Sage leverage highly specialized packages included in Sage that are written and vetted by experts. Many of these packages have been developed over decades and are reliable and fast.

Users may interact with Sage via a command-line interface or via a web interface known as the Sage Notebook. The Sage Notebook is a web application that runs in a web browser and serves as a front end for a Sage server, which may run on a remote computer or on the local computer. The online notebook is the easiest way to begin working with Sage since it works in any modern web browser (including on mobile devices) and requires no extra installation. A free public Sage server is accessible from `http://sagemath.org`.

Sage understands mathematical objects like rings, fields, vector spaces, matrices over different fields, etc. Each vector and matrix knows the ring in which its entries live, such as integers, rationals, elements of a finite field, floating-point double-precision reals (or complexes), high-precision reals (or complexes), or more exotic objects such as multivariate polynomials or algebraic numbers. Different routines or libraries will be transparently used depending on the type of entries. Sage includes many linear algebra libraries which it uses to do computations, including LAPACK, LinBox, IML, M4RI, etc. Support for floating-point double-precision matrices is provided by the included SciPy and NumPy scientific computation packages, which ultimately rely on LAPACK and other standard packages.

Extensive help and additional resources can be obtained freely through the Sage web site at `http://sagemath.org`. Context-sensitive help can be easily obtained by evaluating a function or method name followed by a question mark. Functions and methods can be discovered using tab-completion (see below).

In this chapter, example lines that start with `sage:` are input lines, and everything on an input line following the `sage:` prompt should be typed into Sage. Evaluate input by pressing the Enter key (in the command line) or by clicking evaluate or pressing Shift-Enter (in the notebook). In the examples below, the Sage output immediately follows the input on separate lines without `sage:` prompts. An ellipsis `...` indicates output that has been deleted for brevity.

91.2 Working with Sage

Facts:

1. You can use Sage in a web browser by accessing the Sage Notebook, which can be served from your own computer or a remote computer (such as a public server accessible from `http://sagemath.org`). You can also run Sage in command-line mode on your own computer (`exit` or `quit` will stop Sage). If you are running Sage in command-line mode, you can start the online notebook server for only your own computer by typing `notebook()` at the `sage:` prompt; stop the notebook with Control-C. You can also download an Apple OS X Sage application that lets you start a local Sage notebook or command-line environment by double-clicking.
2. Users use the general-purpose, popular, and easy-to-learn Python programming language to interact with Sage. Python is used in many scientific and other disciplines, so learning Python has many applications beyond Sage. Sage also includes many libraries that are compiled C, C++, Fortran, or other languages, but the "glue" tying everything together, as well as a substantial mathematical library, is also implemented in Python.
3. Mathematical objects in Sage belong to "parents," an idea pioneered by the Magma computer algebra system. Allowable operations, available methods, implemented algorithms, and documentation are all sensitive to an object's parent.
4. What Sage prints (the text representation), and what Sage knows, are two different things, and often what is printed is not the full story. The `parent()` method is helpful in understanding an object.

5. Much of Sage is object-oriented, and vectors, matrices, and vector spaces are considered objects. If an object is assigned a name, then typing the name, then a period, and then a method name followed by parentheses will have Sage call the method to return a result. For example, if `A` is a matrix, then `A.det()` will calculate the determinant of `A` using the `det()` method. Typing the name, appending a period, and then pressing the `Tab` key will bring up a full list of the methods available for that object (e.g., if `A` is a matrix, try `A.<tab>`, or `A.a<tab>` to see all methods beginning with an "a"). Many computations are also available as top-level commands (you can use either `det(A)` or `A.det()` to calculate the determinant). Appending a question mark to a command or method name, such as `A.det?`, and evaluating gives the documentation for the function or method (and the method documentation is context sensitive, depending on the object). Appending two question marks will display the source code for the command, method, or object.
   Documentation of commands and methods contains short informative examples that are tested for accuracy. The possible methods and documentation for vectors and matrices may greatly depend on the types of entries it contains. A quick way to learn about the capabilities of Sage is to build the revelant object (for example, a matrix) and explore methods and their documentation using tab completion and question marks.
6. Sage implements a variety of rings and fields that can be used as entries of vectors or matrices. Some common fields are listed below. The `RDF`, `CDF`, `RR`, and `CC` fields have 53-bit precision (see Section 91.11). `RealField`, `ComplexField`, `RealIntervalField`, and `ComplexIntervalField` support arbitrary precision. All other fields listed are exact with no round-off error.

Field	Name	Notes
Symbolic	`SR`	Used with symbolic variables
Rationals	`QQ`	
Mod p	`Integers(p)` or `GF(p)`	p prime, $p = 2$ highly optimized
Finite Field	`GF(p^n, 'a')`	p prime
Q[sqrt(d)]	`QuadraticField(d, 'a')`	Generator `a`
Cyclotomic Fields	`CyclotomicField(n)`	Rationals with nth roots of unity
Number Fields	`NumberField(poly, 'a')`	Irreducible `poly`, generator `a`
Algebraic Numbers	`QQbar`	Algebraic closure of `QQ`
Algebraic Reals	`AA`	Real numbers in `QQbar`
Machine Reals	`RDF`	Best for numerical linear algebra
Machine Complexes	`CDF`	Best for numerical linear algebra
Reals	`RealField(prec)`	`prec` = precison in bits
Complexes	`ComplexField(prec)`	`prec` = precison in bits
Reals	`RR`	Same as `RealField(53)`
Complexes	`CC`	Same as `ComplexField(53)`
Real Interval	`RealIntervalField(prec)`	`prec` = precision in bits
Complex Interval	`ComplexIntervalField(prec)`	`prec` = precision in bits

7. Create symbolic variables with `var()`. Note that `x` is predefined at startup to be a symbolic variable (you will need to create any other symbolic variables).
8. Indices of vectors, matrix rows, matrix columns, lists, and tuples all begin at 0. For example, the third row of a matrix is accessed with the index 2. While perhaps uncomfortable at first, this makes programming easier, and makes vectors and matrices consistent with similar Python constructs.
9. Large matrices (at least 20 rows or 50 columns) do not print all their entries by default. To print the entries, do `print A.str()`. You can adjust the cutoff for this behavior using `sage.matrix.matrix0.set_max_rows` and `sage.matrix.matrix0.set_max_cols`.

10. We can save vectors and matrices (and most objects in Sage) to a file with the `save()` method. Given a filename, the `load()` function returns the saved object. This lets us save the results of a computation and then resume it later or on another computer.
11. Vectors and matrices may be immutable, which means that they cannot be changed. For example, the reduced row-echelon form of a matrix is typically cached (and hence made immutable) so that another computation for the same matrix will not need to repeat the reduction to echelon form. We can make a matrix `A` immutable using `A.set_immutable()`. We can use the `copy()` function to make a mutable copy.
12. Place comments in your code by prefixing each comment with `#`. Comments extend from `#` to the end of the line.
13. Place several commands on a single line by separating them with a semi-colon: `a = 4; a + 7`. Several quantities can be output in one tuple by creating a single command with values separated by commas: `2+3, 8-7`.
14. In the notebook, only the value of the last expression is printed. An assignment of a value to a variable will not produce any output. A common idiom for assigning `v` to the result of a command and then displaying `v` is `v = some_command(); v`.
15. Lists in Sage are delimited by brackets `[,]`. Entries may be added, removed, and sorted. Duplicates are allowed. Sage also has a `set` data type, for which duplicates are removed, and determining membership is very fast. Tuples are very similar to lists, but are immutable, and are delimited by parentheses `(,)`. Be careful not to confuse tuples with vectors, since vectors also print using parentheses. Lists, tuples, and sets are part of the Python language. The official Python tutorial in the Python documentation has an excellent discussion of lists, tuples, and sets.
16. Python has a very convenient way to build lists using list comprehensions, which is similar to set-builder notation.
17. `lambda` is a reserved word in Python, so it cannot be used as a variable name! `lambda` is used to create short unnamed functions (i.e., "anonymous" functions).
18. By default, `I` and `i` are both predefined as the complex number $i = \sqrt{-1}$.
19. Occasionally, you may want to overwrite a variable that Sage predefines with your own value, like using `i` as an index or setting `I` to be an identity matrix. To access Sage's predefined variable, prepend `sage.all.` to the variable (e.g., `sage.all.I` is the square root of -1, even if you have redefined `I`). Use the `reset()` function to reset variables to Sage defaults (e.g., `reset('I')` resets `I` to $\sqrt{-1}$, and `reset()` will reset all variables to Sage's default values).

Examples:

1. The parent of an object helps distinguish between objects that may print the same.

```
sage: 5, parent(5) # this "5" is an integer
(5, Integer Ring)
sage: 5/1, parent(5/1) # this "5" is a rational
(5, Rational Field)
```

2. Mutable and immutable matrices:

```
sage: A = matrix(QQ, 3, 4, range(12)); A.is_mutable()  # can change A
True
sage: B = A.rref(); B.is_immutable()  # can't change B
True
sage: C = copy(B); C.is_mutable()  # can change C
True
sage: C[0,0] = 100  # change the upper-left element of C
```

3. List comprehensions provide an easy way to map functions or filter.

```
sage: v = vector([-2, 3, 4, -6, 5])
sage: [i^2 for i in v], [i^2 for i in v if i > 0]
([4, 9, 16, 36, 25], [9, 16, 25])
```

91.3 Vectors

Commands:

1. **Constructors**: Sage has a number of ways to construct vectors.
   (a) Construct vectors over a specific ring with `vector(ring, entries)`.
   (b) The ring is optional; if not provided, Sage will infer the ring from the entries. The `base_ring()` method gives the ring of the entries. We can construct a new vector over a different base ring by using the `change_ring()` method.
   (c) Create vector-valued symbolic functions with `f(inputs) = [outputs]` syntax.
   (d) The `zero_vector()` and `random_vector()` commands construct zero and random vectors. The options to `random_vector()` depend on the ring; see the `random_element()` method documentation for the ring for details (for example, `QQ.random_element?`).
2. **Properties**
   (a) In Sage, vectors do not have a column or row orientation. Instead, Sage interprets the vector as either a row or column as the situation demands. In order to explicitly get a row or column vector (i.e., a single-row or single-column matrix), use the `row()` or `column()` methods.
   (b) Use square brackets to access an element of the vector. Remember that the indices start at zero.
   (c) To get the number of elements in a vector, use the `len()` command.
   (d) The `norm()` method gives the vector 2-norm. We can specify a p to compute a p-norm for any $p \geq 1$ (including `Infinity`).
   (e) The `n()` method gives a numerical approximation. We can specify a precision or number of digits.
3. **Manipulations**
   (a) Linear combinations use standard notation, like `2*u + 3*v`.
   (b) The default product `u*v` of two vectors is the dot product. We can also use the `dot_product()` method. The `hermitian_inner_product()` method conjugates the first vector before taking the dot product. There are also `outer_product()` and `cross_product()` methods (cross product works for 3- and 7-dimensional vectors).
   (c) The `apply_map()` method will apply a function to each element of the vector. Equivalently, you could also explicitly construct a new vector using a list comprehension.

Examples:

1. Construct vectors:

```
sage: v = vector(QQ, [1,2,3]); v
(1, 2, 3)
sage: v.change_ring(RDF)
(1.0, 2.0, 3.0)
sage: v.apply_map(lambda x: x^2)  # also f(x) = x^2; v.apply_map(f)
(1, 4, 9)
sage: random_vector(QQ, 10, num_bound=15, den_bound=5)  # random
(-2, -5/2, 15/4, 1, -11/4, 1, -13/4, 10, -3, 1)

sage: x,y = var('x,y'); vector(SR, [x, y, x*sin(y)])
(x, y, x*sin(y))
sage: f(x,y) = [x, y, x*sin(y)]; f  # vector-valued function
(x, y) |--> (x, y, x*sin(y))
sage: f(1,2)
(1, 2, sin(2))
```

2. Properties of a vector. `w.n(20)` gives **w** with 20 bits of precision; use `w.n(digits=20)` for 20 digits.

```
sage: v = vector(QQ, [21,-3,-1,2]); v[0], len(v)
(21, 4)
sage: v.norm(), v.norm(1), v.norm(2), v.norm(5), v.norm(Infinity), v.norm(x)
(sqrt(455), 27, sqrt(455), 4084377^(1/5), 21, (2^x + 3^x + 21^x + 1)^(1/x))
sage: w = vector(SR, [pi, e, 3/7]); w.n(), w.n(digits=2)
((3.14159265358979, 2.71828182845905, 0.428571428571429), (3.1, 2.7, 0.43))
```

3. Vector arithmetic:

```
sage: u = vector(QQ, [1, 2, 3]); v = vector(QQ, [4, 3, -1])
sage: 2*u + 3*v, u.cross_product(v)
((14, 13, 3), (-11, 13, -5))
sage: u*v, u.dot_product(v), vector(SR, [4,2+I,3]).hermitian_inner_product(u)
(7, 7, -2*I + 17)
sage: u.outer_product(v)  # same as: u.column() * v.row()
[ 4  3 -1]
[ 8  6 -2]
[12  9 -3]
```

91.4 Matrices

Commands:

1. **Constructors**
   (a) The `matrix(ring, entries)` command constructs a matrix over a specific ring of a specific size. The entries can be given as a list of rows or row vectors. If the dimensions of the matrix are supplied before the entries parameter, then the entries parameter can just be a list of entries or a function mapping indices (zero-based) to entry values. A base ring will be inferred if not specified.

(b) The `column_matrix()` command takes a list of columns instead of a list of rows.

(c) A variety of functions construct special types of matrices.

Description	Example
Diagonal matrix	`diagonal_matrix(QQ, [1,2,3])`
Identity matrix	`identity_matrix(3)`
Zero matrix	`zero_matrix(3,4)`
All-ones matrix	`ones_matrix(3,4)`
Elementary matrix	`elementary_matrix(QQ, 4, row1=3, scale=-2)`
Random matrix	`random_matrix(RDF, 3,4)`
Hadamard matrix	`hadamard_matrix(4)`
Polynomial companion matrix	`companion_matrix((x^2-2*x+1).polynomial(QQ))`

The `zero_matrix()`, `ones_matrix()`, and `random_matrix()` functions also can take a single number to construct a square matrix. For more options to give to the `random_matrix()` function, see the `random_element()` method for the ring of entries (e.g., `RDF.random_element?`). The `random_matrix()` constructor also takes an `algorithm` keyword that can be used to create small example matrices with integer entries that are "nice" to work with. Values of this keyword can be `echelonizable`, `unimodular`, `subspaces`, or `diagonalizable` and will create matrices (respectively) with a "nice" echelon form; with determinant one; with simultaneously "nice" bases for the row space, column space, right kernel, and left kernel; or similar to a diagonal matrix.

(d) The included SciPy linear algebra package has a number of matrices as well, including the above and Hankel, Leslie, Pascal, Toeplitz matrices, and more.

(e) You can construct the space of all matrices `M` with a given base ring and size and use this space to construct matrices. There are also other methods of matrix spaces; see the Sage reference manual.

2. **Block Matrix Constructors**

(a) The `block_matrix()` and `block_diagonal_matrix()` commands create block matrices (see Example 2 next). Blocks may be rectangular, integer arguments are converted to blocks with diagonal entries equal to the integer, and the result carries the natural subdivisions by default. See the `block_matrix()` documentation for many more features.

(b) The matrix methods `block_sum()` (for block diagonal results), `augment()` (concatenate horizontally), and `stack()` (concatenate vertically) provide alternative ways to combine two matrices or a matrix and a vector.

(c) Sage matrices can be subdivided into blocks (i.e., partitioned into submatrices). Blocks are created with the `subdivide()` method (specify row and column indices before each subdivision point), while the `subdivisions()` method returns this subdivision. The `subdivision()` method retrieves a specified block, and the `subdivision_entry()` returns a specified entry of a specified block. If arithmetic is done between matrices with compatible subdivisions, the result also has the natural subdivision. See Example 2 next.

3. **Properties**

(a) The `I`, `C`, `T`, and `H` attributes give the inverse, conjugate, transpose, and conjugate transpose (Hermitian transpose) of the matrix. Do not use parentheses when using these (e.g., `A.I`). The `I`, `C`, `T`, and `H` attributes are shortcuts for the `inverse()`, `conjugate()`, `transpose()`, and `conjugate_transpose()` methods, respectively. The inverse of a matrix is also `A^-1` and `~A`.

(b) Matrices have a number of methods for standard computations (e.g., `A.det()`).

Method	Description
`density()`	Density of nonzero entries
`det()`	Determinant
`minors(k)`	List of all k by k minors, in lexicographic order
`ncols()`	Number of columns
`norm(p)`	Norm, `p` can be 1, 2, `Infinity`, or `'frob'` (Frobenius)
`nrows()`	Number of rows
`permanent()`	Permanent
`permanental_minor(k)`	Sum of permanents of all possible k by k submatrices
`rank()`	Rank
`trace()`	Trace

(c) There are also several other commands to return various matrices.

Description	Example
Adjoint	`A.adjoint()`
Antitranspose	`A.antitranspose()`
Commutator ($AB - BA$)	`A.commutator(B)`
Matrix exponential, $\sum_{k=0}^{\infty} \frac{A^k}{k!}$	`A.exp()` (Use `RDF` or `CDF` matrices)
Elements satisfying given condition	`A.find(lambda x: x<0)`
Numerical approximation	`A.n()`, `A.n(digits=4)`, `A.n(10)` (10 bits)

The `find()` method makes a matrix with ones where the original matrix entries satisfied the given condition and zeros elsewhere. The `indices=True` option returns instead a dictionary of indices and elements satisfying the condition.

(d) The `A.iterates(v, n, rows=...)` method takes a vector $\mathbf{v}$ and a number n, and returns the vectors $\mathbf{v}, A\mathbf{v}, A^2\mathbf{v}, \ldots, A^{n-1}\mathbf{v}$ as rows (if `rows=True`, which is default) or columns (if `rows=False`) of a matrix.

(e) The `charpoly()` and `minpoly()` methods return the characteristic and minimal polynomial, respectively. Each function optionally takes a variable name.

(f) The following methods give the indices of various nonzero elements of a matrix, row, or column (remember that indices start at zero):
`nonzero_positions()`, `nonzero_positions_in_row()`, and `nonzero_positions_in_column()`.

(g) `A.column_space()`, `A.row_space()`, `A.right_kernel`, and `A.left_kernel()` construct the corresponding vector spaces. *Warning*: Default versions of these commands are generally the left versions, so `A.kernel()` is the left kernel and `A.image()` is the row space, consistent with the $\mathbf{x} \mapsto \mathbf{x}A$ transformation, where the vector is on the left in the product.

(h) Matrices have numerous predicates that start with `is_`, for example, `A.is_invertible()`. The `is_similar()` method can optionally return a transformation matrix.

Method	True if
`is_bistochastic()`	Each row and column sums to 1
`is_dense()`	Matrix is stored in a dense data structure
`is_diagonalizable()`	Matrix is similar to a diagonal matrix
`is_hermitian()`	Matrix is equal to its conjugate transpose
`is_idempotent()`	Matrix is equal to its square
`is_immutable()`	Matrix entries cannot be changed
`is_invertible()`	Matrix is invertible over its base ring
`is_mutable()`	Matrix entries can be changed
`is_one()`	Matrix is identity matrix
`is_scalar()`	Matrix is a multiple of the identity matrix
`is_similar()`	Matrix is similar to another given matrix
`is_singular()`	Matrix is not invertible
`is_skew_symmetric()`	Matrix is equal to its negative transpose
`is_sparse()`	Matrix is stored in a sparse data structure
`is_square()`	Matrix has the same number of rows and columns
`is_symmetric()`	Matrix is equal to its transpose
`is_unitary()`	Columns form an orthonormal basis
`is_zero()`	Every entry is zero

4. **General manipulations**
   (a) Arithmetic with matrices uses standard notation, like `2*A - A*B + A^3 - A^-1`.
   (b) Recall that Sage vectors do not have an implicit row or column orientation. If A is a matrix and **v** is a vector, then `A*v` views **v** as a column vector and `v*A` views **v** as a row vector. In order to get a specific orientation for **v**, convert it to a single-row or single-column matrix using `v.row()` or `v.column()`.
   (c) `A.elementwise_product(B)` computes the Hadamard product of A and B, while `A.tensor_product(B)` computes the tensor product. Subdivisions are automatically added for tensor products; do `A.tensor_product(B, subdivide=False)` to not have the result subdivided. `A.trace_of_product(B)` method computes the trace of AB without actually computing AB.
   (d) `A.apply_map(function)` will return a new matrix resulting from applying the function to each element of A, like `f(x) = x^2; A.apply_map(f)` or simply `A.apply_map(lambda x: x^2)`.
5. **RREF**: There are several ways to compute reduced row-echelon form (e.g., `A.rref()`).

Method	Description
`rref()`	Calculate RREF (over fraction field)
`echelon_form()`	Calculate RREF (over base ring)
`echelonize()`	Modify the matrix to echelon form (over base ring)
`extended_echelon_form()`	Augment with identity matrix before reducing

   (a) There are a number of commands to perform elementary operations on a matrix. Each command also has a variant that returns a new matrix instead of modifying the matrix—these variants have the same name, but in past tense and prefixed by `with_`. For example, `A.rescale_row(0,2)` rescales row 0 of A by a factor of 2, while `A.with_rescaled_row(0,2)` does not modify A, but returns a new matrix that is equal to A, except row 0 is scaled by 2. Remember again that row and column indexing starts at zero.

Row operation	Example
$C_2 \leftarrow C_2 - 4C_1$	`A.add_multiple_of_column(2,1,-4)`
$R_2 \leftarrow R_2 - 4R_1$	`A.add_multiple_of_row(2,1,-4)`
$C_0 \leftarrow 3C_0$	`A.rescale_col(0, 3)`
$R_0 \leftarrow 3R_0$	`A.rescale_row(0, 3)`
$C_0 \leftarrow 3C_2$	`A.set_col_to_multiple_of_col(0,2,3)`
$R_0 \leftarrow 3R_2$	`A.set_row_to_multiple_of_col(0,2,3)`
$C_0 \leftrightarrow C_2$	`A.swap_columns(0,2)`
$R_0 \leftrightarrow R_2$	`A.swap_rows(0,2)`

(b) The `pivots()` method returns the indices of the pivot columns and `nonpivots()` returns the indices of the nonpivot columns. The `pivot_rows()` method returns the indices of a topmost subset of the rows that span the row space and are linearly independent.

6. **Solving systems**: Sage has specialized methods for solving systems of linear equations over a variety of rings, which can be significantly more efficient and stable than inverting a matrix. To solve a system of linear equations of the form $A\mathbf{x} = \mathbf{b}$ or $AX = B$ (X and B matrices), use the matrix `solve_right()` method or the synonymous backslash syntax `A \ b`. The `solve_left()` method solves equations of the form $\mathbf{x}A = \mathbf{b}$ or $XA = B$.
7. **Left vs. Right**: Sage can do many calculations dealing with right ($A\mathbf{x}$) or left ($\mathbf{x}A$) matrix-vector products. The right or left tells which side the vector is on. Typically, "right" computations return vectors of interest as columns, while "left" computations return vectors of interest as rows. *Warning*: Sage typically defaults to left versions of commands (e.g., `A.kernel()` is `A.left_kernel()`). Use the right or left versions of commands below to be explicit.

Right: $A\mathbf{x} = \mathbf{b}$	Left: $\mathbf{x}A = \mathbf{b}$
`right_eigenmatrix()`	`left_eigenmatrix()`
`right_eigenspaces()`	`left_eigenspaces()`
`right_eigenvectors()`	`left_eigenvectors()`
`right_kernel()`	`left_kernel()`
`right_nullity()`	`left_nullity()`
`solve_right()`	`solve_left()`

8. **Indexing**: Sage supports very flexible ways of getting and setting elements of matrices. `A[i,j]` returns the entry in row i and column j (indices start at zero). The following table gives some general patterns for the index syntax, which is similar to NumPy or MATLAB syntax. If an index is negative, it counts backward from the last index (e.g., `A[-1,:]` is the last row), and if a step size is negative, it indicates a count down (e.g., `A[::-1,:]` gives a new matrix with the rows reversed). See the Sage reference manual on matrix indexing, `http://www.sagemath.org/doc/reference/sage/matrix/docs.html#indexing`.

Index	Description
`i`	Index i
`:`	All indices
`i:`	Indices from i to the end
`:j`	Indices up to, but not including, j
`i:j`	Indices from i up to, but not including, j
`i:j:s`	Every sth index from i up to, but not including, j
negative	Count backwards from the last index
list	Explicit list of indices, possibly with repeats and reorderings

In the methods below, row and column index lists can contain repeated or reordered indices. The row and column methods at the end of the table below return vectors or lists of vectors, while the equivalent index notation returns submatrices. The second table below indicates how to modify submatrices using methods or indexing notation.

Method	Equivalent indexing
`A.diagonal()`	None
`A.matrix_from_columns([2,3])`	`A[:,[2,3]]`
`A.matrix_from_rows([2,3])`	`A[[2,3],:]`
`A.matrix_from_rows_and_columns([2,3],[3,4])`	`A[[2,3],[3,4]]`
`A.submatrix(i,j,nrows,ncols)`	`A[i:i+nrows, j:j+ncols]`
`A.column(i)`	`A[:,i]`
`A.columns([2,3])`	`A[:[2,3]]`
`A.row(i)`	`A[i,:]`
`A.rows([2,3])`	`A[[2,3],:]`

Method	Equivalent indexing
`A.set_block(i,j,B)`	`A[i:i+B.nrows(), j:j+B.ncols()]=B` (B a matrix)
`A.set_column(i, [2,1,3])`	`A[:,i]=vector([2,1,3])` or `A[:,i]=[[2],[1],[3]]`
`A.set_row(i, [2,1,3])`	`A[i,:]=vector([2,1,3])` or `A[i,:]=[[2,1,3]]`

9. **Sparse Matrices**: In order to handle large sparse matrices, Sage can store any matrix in a compressed way so that only the nonzero entries are stored. Create a sparse matrix by specifying `sparse=True` when the matrix is created. Matrices initialized from a Python dictionary default to sparse. The included SciPy Python package also has many resources for dealing with sparse matrices.
10. **Decompositions**: A variety of matrix decompositions are available in Sage, and many are available with algorithms for both exact and numerical matrices. Typically the return value is a tuple of matrices that can be used to reconstruct the matrix (except Jordan form, which requires explicitly asking for the transformation matrix). Some exact algorithms require the base ring of the matrix to have certain properties, such as containing square roots. In other cases, the matrix must have certain properties, such as having a characteristic polynomial that factors. These conditions are listed in the "Base ring must contain" column.

    Sage implements the algebraically closed field of algebraic numbers, `QQbar`, so some decompositions applied to matrices of rational numbers will automatically return matrices with entries from `QQbar` as necessary (e.g., Cholesky). On the other hand, `jordan_form` will simply fail if there are eigenvalues outside the base ring.

    Matrices over `RDF` and `CDF` use specialized numerical algorithms for the LU, QR, SVD, Schur, and Cholesky factorizations; see Section 91.11.

Name	Method	Base ring must contain
LU, Triangular	`LU()`	Fraction
QR, Gram-Schmidt	`QR()`	Fraction, square roots
SVD, Singular Value	`SVD()`	
Schur	`schur()`	
Cholesky, Square Root	`cholesky()`	Square roots
Jordan Form	`jordan_form()`	Eigenvalues
Rational Canonical Form (Invariant Factors)	`rational_form()`	Field
Smith Form	`smith_form()`	Principal ideal domain
Symplectic Form	`symplectic_form()`	Field
Subspace Decomposition	`decomposition()`	Factored characteristic polynomial

Examples:

1. Constructing matrices.

```
sage: matrix(QQ, [[1,0,1],[2,0,-1]])  # also matrix(QQ, 2, 3, [1,0,1,2,0,-1])
[ 1  0  1]
[ 2  0 -1]
sage: u = vector(QQ, [2,3]); v = vector(QQ, [5,0]); w = vector(QQ, [-1,2])
sage: column_matrix([u,v,w])
[ 2  5 -1]
[ 3  0  2]
sage: def vandermonde(R, v): return matrix(R, len(v), lambda i,j: v[i]^j)
sage: vandermonde(QQ, [2,3,4])
[ 1  2  4]
[ 1  3  9]
[ 1  4 16]
sage: def hilbert(R,n): return matrix(R, n, lambda i,j: 1/(i+j+1))
sage: hilbert(QQ, 3)
[  1 1/2 1/3]
[1/2 1/3 1/4]
[1/3 1/4 1/5]
```

2. Constructing block matrices.

```
sage: A = matrix(QQ, [[-1,1],[0,-1]]);  B = matrix(QQ, [[1,-1/2]])
sage: block_matrix(QQ, [[A,1],[0,B]])
[  -1    1|   1    0]
[   0   -1|   0    1]
[---------+---------]
[   0    0|   1 -1/2]
sage: block_diagonal_matrix([A,B], subdivide=False)
[  -1    1    0    0]
[   0   -1    0    0]
[   0    0    1 -1/2]
sage: A.augment(vector(QQ, [1,-1/2]), subdivide=True)
[  -1    1|   1]
[   0   -1|-1/2]
```

```
sage: A.stack(B)
[  -1    1]
[   0   -1]
[   1 -1/2]
```

We can subdivide an existing matrix to get a block matrix. Here, we subdivide just before row 2 and just before columns 1 and 3 (remember indices start at 0).

```
sage: C = matrix(QQ, 3, 5, range(15))
sage: C.subdivide([2],[1,3]); C
[ 0| 1  2| 3  4]
[ 5| 6  7| 8  9]
[--+-----+-----]
[10|11 12|13 14]
sage: C.subdivision(0,2) # block (0,2)
[3 4]
[8 9]
sage: C.subdivision_entry(0,2,0,1) # entry (0,1) in block (0,2)
4
sage: C * C.transpose() # arithmetic preserves block structure if possible
[ 30  80|130]
[ 80 255|430]
[-------+---]
[130 430|730]
```

3. Properties of a matrix.

```
sage: A = matrix(QQ, 2, [0,-1,2,-2]); A, A.I, A.T, A.find(lambda x: x<0)
(
[ 0 -1]  [ -1 1/2]  [ 0  2]  [0 1]
[ 2 -2], [ -1   0], [-1 -2], [0 1]
)
sage: A.find(lambda x: x<0, indices=True)
{(0, 1): -1, (1, 1): -2}
sage: A.change_ring(RDF).exp() # use RDF for numeric calculations
[    0.508325986 -0.309559875653]
[ 0.619119751306 -0.110793765307]
sage: A.norm(), A.norm(1), A.norm(Infinity), A.norm('frob')
(2.92080962648, 3.0, 4.0, 3.0)
```

```
sage: A = diagonal_matrix([2,2,5])
sage: A.charpoly(), A.minpoly()
(x^3 - 9*x^2 + 24*x - 20, x^2 - 7*x + 10)
sage: A.minors(2)
[4, 0, 0, 0, 10, 0, 0, 0, 10]
sage: A.det(), A.rank(), A.trace(), A.permanent(), A.permanental_minor(2)
(20, 3, 9, 20, 24)
```

4. Like many computer algebra systems, row-reducing a symbolic matrix using the `rref()` method assumes that we are working over a field (i.e., that if `c` is any nonzero element of the base ring, then `1/c` is also in the base ring). Since Sage can also work with matrices over general rings (which may not be fields), we can ask Sage to do operations without assuming that we can divide. For example, if a matrix has entries in a polynomial ring, we can ask for the `echelon_form()` of the matrix, which only uses operations in the polynomial ring (e.g.,

Sage will not divide except by constants). We can use the `matrix_over_field()` method to get a copy of a matrix over the fraction field of its base ring.

```
sage: R.<c1,c2,c3> = QQ[] # construct a polynomial ring with 3 variables
sage: A = matrix([[1,1,2,c1], [1,0,1,c2], [2,1,3,c3]]); A
[ 1  1  2 c1]
[ 1  0  1 c2]
[ 2  1  3 c3]
sage: A.rref() # works over fraction field of R, so 1/c1 exists, etc.
[1 0 1 0]
[0 1 1 0]
[0 0 0 1]
sage: A.echelon_form() # uses only operations in the polynomial ring R
[            1             0             1            c2]
[            0             1             1       c1 - c2]
[            0             0             0 -c1 - c2 + c3]
sage: B = A.matrix_over_field(); B.base_ring()
Fraction Field of Multivariate Polynomial Ring in c1, c2, c3 over Rational Field
sage: B.echelon_form() == A.rref()
True
```

5. We can easily get pivot columns.

```
sage: A = matrix(QQ, [[1,2,3,4],[2,4,6,8],[3,5,4,3]]); A.rref()
[  1   0  -7 -14]
[  0   1   5   9]
[  0   0   0   0]
sage: A.pivots(), A.nonpivots(), A.pivot_rows()
((0, 1), (2, 3), (0, 2))
sage: A[:,A.pivots()]  # get the pivot columns
[1 2]
[2 4]
[3 5]
sage: A[A.pivot_rows(),:]  # get the pivot rows
[1 2 3 4]
[3 5 4 3]
```

6. The `extended_echelon_form()` method first augments an m by n matrix with an m by m identity matrix before computing the rref. The result can also be subdivided by specifying `subdivide=True`. The four relations in the last two lines of this example are true in general.

```
sage: A = matrix(QQ, 3, 3, [2,3,4,-1,2,4,1,5,8])
sage: B = A.extended_echelon_form(subdivide=True); B
[   1    0 -4/7|   0 -5/7  2/7]
[   0    1 12/7|   0  1/7  1/7]
[--------------+--------------]
[   0    0    0|   1    1   -1]
sage: C = B.subdivision(0, 0); C  # C is upper-left submatrix
[   1    0 -4/7]
[   0    1 12/7]
sage: L = B.subdivision(1, 1); L  # L is lower-right submatrix
[ 1  1 -1]
sage: A.right_kernel() == C.right_kernel(), A.row_space() == C.row_space()
(True, True)
sage: A.column_space() == L.right_kernel(), A.left_kernel() == L.row_space()
(True, True)
```

7. We solve systems of equations using `solve_right` or backslash notation.

```
sage: A = matrix(QQ, 2, [1,2,3,4]); b = vector(QQ, [5,6])
sage: x = A.solve_right(b); x  # also x = A \ b
(-4, 9/2)
sage: A * x == b
True
sage: B = matrix(QQ, 2, 3, [5,1,0,6,0,1])
sage: X = A \ B; A * X == B  # also X = A.solve_right(B)
True
```

8. To be able to compute an exact QR decomposition, we need a field that contains square roots, so we create a matrix with entries from `QQbar`. Sage uses specialized numerical algorithms if the matrix is over `RDF`.

```
sage: A = matrix(QQbar, 3, 3, [0..8])  # [0..8] is [0, 1, 2, ..., 8]
sage: Q, R = A.QR(); A == Q*R and Q.is_unitary()  # True if both conditions are True
True
sage: B = matrix(RDF, 3, 3, [0..8])
sage: Q, R = B.QR(); (A - Q*R).norm() < 10^-10 and Q.is_unitary()
True
```

91.5 Eigenvalues and Eigenvectors

Sage will compute exact eigenvalues, eigenvectors, and eigenspaces for matrices with entries from exact rings, and will compute approximate eigenvalues and eigenvectors for matrices with floating-point entries. Both right and left variants of the eigenvector methods are available; we will only discuss the right variants.

Commands:

1. The `eigenvalues()` method computes the eigenvalues. If a matrix over `QQ` does not have rational eigenvalues, then the eigenvalues may be returned as algebraic numbers in `QQbar` (the algebraic completion of `QQ`) and printed as numeric approximations followed by question marks. The question mark indicates that the number is contained in the interval found by taking the last digit of the printed representation plus or minus one (e.g., `3.25?` represents an exact root in the interval $[3.24, 3.26]$). We emphasize that a `QQbar` element is an exact root of a polynomial and the printed approximation indicates an interval containing the value, and so may be slower to work with than computing the eigenvalues numerically using an `RDF` or `CDF` matrix.
2. The `eigenvectors_right()` method returns a list, each element of the following form: `(eigenvalue, list of eigenvectors, algebraic multiplicity)`.
   The `eigenspaces_right()` method returns a list, each element of the following form: `(eigenvalue, eigenspace)`, where the eigenspace is a Sage vector space.
3. The `eigenmatrix_right()` method of a matrix A returns two matrices D and P such that $AP = PD$. The eigenvalues are the diagonal of D and the corresponding eigenvectors are the columns of P. Columns of zeros in P indicate that the geometric multiplicity of the eigenvalue is not equal to the algebraic multiplicity.

Examples:

1. We calculate various eigenvalues, eigenvectors, and eigenspaces. Numerical matrices always claim eigenvalues have multiplicity one since numerical error can easily fool multiplicity calculations (see Section 91.11).

```
sage: entries = [-8,21,74,-48,24,-7,-78,72,-12,6,44,-36,-4,-3,2,-4]
sage: A = matrix(QQ, 4, 4, entries); A.eigenvalues()
[5, 4, 8, 8]
sage: A.eigenvectors_right()
[(5, [ (1, -5/7, 2/7, -1/7) ], 1),
 (4, [ (1, -3/2, 3/4, 1/4) ], 1),
 (8, [ (1, 0, 0, -1/3), (0, 1, -1/2, -1/3) ], 2)]
sage: A.eigenspaces_right()
[
(5, Vector space of degree 4 and dimension 1 over Rational Field ...),
(4, Vector space of degree 4 and dimension 1 over Rational Field ...),
(8, Vector space of degree 4 and dimension 2 over Rational Field ...)
]
```

2. The `eigenmatrix_right()` method provides a very easy way to get corresponding lists of eigenvalues and eigenvectors using the `diagonal()` and `columns()` methods of matrices.

```
sage: entries = [-8,21,74,-48,24,-7,-78,72,-12,6,44,-36,-4,-3,2,-4]
sage: A = matrix(QQ, 4, 4, entries)
sage: D, P = A.eigenmatrix_right(); D, P, A * P == P * D
(
[5 0 0 0]  [   1    1    1    0]
[0 4 0 0]  [-5/7 -3/2    0    1]
[0 0 8 0]  [ 2/7  3/4    0 -1/2]
[0 0 0 8], [-1/7  1/4 -1/3 -1/3], True
)
sage: evals = D.diagonal(); evecs = P.columns()
sage: A*evecs[0] == evals[0] * evecs[0]  # check first eigenvalue/eigenvector
True
```

3. For matrices with rational entries, we can optionally ask that eigenspaces be reported just once per irreducible factor of the characteristic polynomial since the eigenspaces for each irreducible factor are related in a natural way.

```
sage: A = matrix(QQ, 3, 3, [-7, 2, -22, 10, -3, 33, 3, -1, 10]); A.charpoly()
x^3 + 1
sage: A.eigenspaces_right(format='galois')
[
(-1, Vector space of degree 3 and dimension 1 over Rational Field ...),
(a1, Vector space of degree 3 and dimension 1 over
Number Field in a1 with defining polynomial x^2 - x + 1
User basis matrix:
[         1 1/2*a1 - 2       -1/2])
]
```

4. Even if Sage does not directly compute eigenvalues for matrices over some exotic ring, other tools may provide information you desire. For example, the `fcp()` method, which returns the factored characteristic polynomial, is often useful.

```
sage: F.<a> = FiniteField(3^2);
sage: A = matrix(F, 3, 3, [2*a, a, 2*a, 0, 2, a + 2, 2, a, 2*a]); A.fcp()
(x + a + 1) * (x^2 + a*x + 2)
sage: (A - (-a-1)*identity_matrix(3)).right_kernel()  # an eigenspace
Vector space of degree 3 and dimension 1 over Finite Field in a of size 3^2 ...
```

91.6 Vector Spaces

You can use `VectorSpace` objects to work directly with vector spaces.

Definitions:

1. The **basis matrix** of a vector space has the basis vectors of the vector space as its rows. By default, Sage will store and use a canonical **echelonized** basis for a vector space (i.e., the basis matrix will be a matrix in reduced row-echelon form). A **user basis** can also be explicitly used (see below for examples).
2. Vectors in a vector space V over a field F also live in some **ambient vector space** F^n. The degree of V is n. The **dimension** or **rank** of V is the size of the basis. A vector space is **full** if its degree equals its dimension.

Commands:

1. **Constructors**
   (a) The simplest way to create a vector space is to raise a field to a power (in general, raising a ring to a power will construct a free module). We can also use `VectorSpace` directly by providing a field and a dimension.
   (b) The `parent()` method of a vector returns the vector space containing the vector.
   (c) The `span()` command constructs the vector space spanned by a list of vectors.
   (d) Use the `subspace()` method to construct subspaces. The `subspace_with_basis()` method allows specifying a "user basis".
2. **Properties**
   (a) `v in V` tests if the vector **v** is in the vector space V.
   (a) Check if two vector spaces V and W are equal with `V == W`. Check if V is a subspace of W using `V.is_subspace(W)`.
   (b) The basis (canonical basis) of a vector space is returned by the `basis()` (`echelonized_basis()`) method. The corresponding `basis_matrix()` and `echelonized_basis_matrix()` methods return matrices in which the basis vectors are the rows.
   (c) An element of a vector space can be expressed as a linear combination of the basis vectors (which are canonical by default). The scalars (i.e., coordinates) can be returned as a list (using the `coordinates()` method) or as a vector (using the `coordinate_vector()` method).
   (d) The `linear_dependence()` method takes a list of vectors and returns the coefficients of a nontrivial linear combination that equals zero. An empty sequence is returned if the vectors are linearly independent.
3. We can intersect two vector spaces (`U.intersection(V)`), add them (`U + V`), form their direct sum (`U.direct_sum(V)`), and take their quotient (`U / V`).

Examples:

1. Create a vector space, or more generally, a free module like `ZZ^3`.

```
sage: QQ^3
Vector space of dimension 3 over Rational Field
sage: VectorSpace(RDF, 3)
Vector space of dimension 3 over Real Double Field
```

2. Construct subspaces.

```
sage: u = vector(QQ, [1, 2, 3]); v = vector(QQ, [-3, 0, 3])
sage: S = span([u,v]); S  # basis echelonized, not (u,v)
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -1]
[ 0  1  2]
sage: T = S.subspace([u + v]); T  # the basis matrix is canonical
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[ 1 -1 -3]
sage: U = S.subspace_with_basis([u + v]); U  # basis not echelonized
Vector space of degree 3 and dimension 1 over Rational Field
User basis matrix:
[-2 2 6]
```

3. Properties of vector spaces.

```
sage: u = vector(QQ, [1, 2, 3]); v = vector(QQ, [-1, 0, 1])
sage: W = span([u,v]); V = W.ambient_vector_space()
sage: W.degree(), W.dimension(), W.rank(), W.is_full(), V.is_full(), V == QQ^3
(3, 2, 2, False, True, True)
sage: S = span([v]); S.is_subspace(W), S == span([-v])
(True, True)
```

4. Vectors as elements of vector spaces. The `coordinates()` method below returns a Python list, while the `coordinate_vector()` method returns an actual vector.

```
sage: u = vector(QQ, [1, 2, 3]); v = vector(QQ, [-1, 0, 1]); x = 3*u - 7*v
sage: W = span([u,v]); W
Vector space of degree 3 and dimension 2 over Rational Field ...
sage: x in W, vector(QQ, [3, 7, 195]) in W
(True, False)
sage: W.coordinates(x), W.coordinate_vector(x)  # relative to the canonical basis
([10, 6], (10, 6))
sage: X = (QQ^3).subspace_with_basis([u, v]); X
Vector space of degree 3 and dimension 2 over Rational Field
User basis matrix:
[ 1  2  3]
[-1  0  1]
sage: X.coordinate_vector(x)  # relative to the user basis
(3, -7)
sage: X.linear_combination_of_basis([3, -7])
(10, 6, 2)
```

5. Vector space arithmetic.

```
sage: u = vector(QQ, [1,2,2]); v = vector(QQ, [-1,0,4])
sage: w = vector(QQ, [2,1,5])
sage: U = span([u + v, u - 3*v]); W = span([v + w, 5*v - 6*w])
sage: U.intersection(W)
Vector space of degree 3 and dimension 1 over Rational Field ...
sage: U + W
Vector space of degree 3 and dimension 3 over Rational Field ...
sage: U.direct_sum(W)
Vector space of degree 6 and dimension 4 over Rational Field ...
```

```
sage: u = vector(QQ, [1,2,2,5,3]); v = vector(QQ, [-1,0,4,3,1])
sage: U = span([u, v]); Q = (QQ^5).quotient(U); Q.dimension()
3
sage: phi = Q.quotient_map(); phi
Vector space morphism ...
sage: phi.kernel() == U
True
```

91.7 Linear Transformations

Commands:

1. If A is a matrix, `linear_transformation(A)` constructs the transformation $\mathbf{x} \mapsto \mathbf{x}A$. *Warning*: Notice again the preference for a product with the vector on the left. The domain and codomain are inferred from the base ring and matrix dimensions. Create the linear transformation $\mathbf{x} \mapsto \mathbf{A}\mathbf{x}$ by `linear_transformation(A, side='right')`, which just transposes A to create a left-sided version; the matrix stored and printed by the linear transformation is understood as the left version.
2. We can specify the rule for a linear transformation using a lambda function, a Python function defined with the `def` keyword, a symbolic function, or by a list of images for the elements of the domain's basis in place of the matrix A.
3. If T is a linear transformation and $\mathbf{v}$ is a vector, `T(v)` computes the image of $\mathbf{v}$.
4. `T.is_surjective()` and `T.is_injective()` query properties of a transformation T.
5. `T.image()`, `T.kernel()`, and `T.inverse_image(V)` construct related vector spaces.
6. Compose linear transformations by multiplying them together.
7. `T.restrict_domain()` and `T.restrict_codomain()` construct new transformations.

Examples:

1. Construct some linear transformations.

```
sage: A = matrix(QQ, 2, 3, range(6)); A
[0 1 2]
[3 4 5]
sage: T = linear_transformation(A); T
Vector space morphism represented by the matrix:
[0 1 2]
[3 4 5]
```

```
Domain: Vector space of dimension 2 over Rational Field
Codomain: Vector space of dimension 3 over Rational Field
sage: U = linear_transformation(A, side='right'); U
Vector space morphism represented by the matrix:
[0 3]
[1 4]
[2 5]
Domain: Vector space of dimension 3 over Rational Field
Codomain: Vector space of dimension 2 over Rational Field
sage: v = vector(QQ, [2,3,1]); U(v) == A * v
True
```

We can give the domain and codomain explicitly. The matrix is interpreted (and printed) as the representation relative to the bases of the domain and codomain.

```
sage: V = (QQ^2).subspace_with_basis([vector(QQ, [-1, 1])])
sage: W = (QQ^3).subspace_with_basis([vector(QQ, [0,1,0]), vector(QQ, [0,0,1])])
sage: A = matrix(QQ, [[2,5]]); T = linear_transformation(V, W, A); T
Vector space morphism represented by the matrix:
[2 5]
Domain: Vector space of degree 2 and dimension 1 over Rational Field ...
Codomain: Vector space of degree 3 and dimension 2 over Rational Field
User basis matrix:
[0 1 0]
[0 0 1]
sage: v = vector(QQ, [-4, 4]); T(v)
(0, 8, 20)
sage: V.coordinate_vector(v) * T.matrix() * W.basis_matrix()  # check
(0, 8, 20)
```

The third argument may be a function or list of images instead of a matrix.

```
sage: f = lambda x: vector(QQ, [2*x[0] + x[2], 5*x[1] - 6*x[2]])
sage: T = linear_transformation(QQ^3, QQ^2, f); T
Vector space morphism represented by the matrix:
[ 2  0]
[ 0  5]
[ 1 -6]
Domain: Vector space of dimension 3 over Rational Field
Codomain: Vector space of dimension 2 over Rational Field
```

```
sage: images = [vector(QQ, [1,2]), vector(QQ, [-2,4])]
sage: T = linear_transformation(QQ^2, QQ^2, images); T
Vector space morphism represented by the matrix:
[ 1  2]
[-2  4]
Domain: Vector space of dimension 2 over Rational Field
Codomain: Vector space of dimension 2 over Rational Field
```

2. Query properties of a linear transformation.

```
sage: A = matrix(QQ, 2, 3, range(6))
sage: T = linear_transformation(A)
sage: T.is_surjective(), T.is_injective(), T.image() == T(QQ^2)
(False, True, True)
sage: T.image()
Vector space of degree 3 and dimension 2 over Rational Field ...
sage: T.kernel()
```

```
Vector space of degree 2 and dimension 0 over Rational Field ...
sage: W = (QQ^3).subspace([vector(QQ, [2,3,4])]); T.inverse_image(W)
Vector space of degree 2 and dimension 1 over Rational Field ...
```

3. Composing and restricting linear transformations.

```
sage: A = matrix(QQ, [[2, 4, 2], [3, 6, 3]])
sage: T = linear_transformation(A); v = vector(QQ, [10, -15])
sage: T(v)
(-25, -50, -25)
sage: U = linear_transformation(diagonal_matrix([1, 1/2, 1/5]))
sage: (U * T)(v)
(-25, -25, -5)
```

```
sage: new_domain = (QQ^2).subspace([vector(QQ, [2,-3])])
sage: S = T.restrict_domain(new_domain); S
Vector space morphism represented by the matrix:
[-5/2   -5 -5/2]
Domain: Vector space of degree 2 and dimension 1 over Rational Field ...
Codomain: Vector space of dimension 3 over Rational Field
sage: S(vector(QQ, [10, -15]))
(-25, -50, -25)
```

Note that the matrix representations of the restrictions are with respect to the echelonized bases of the new subspaces.

91.8 Graphics

Sage has extensive capabilities for drawing 2D and 3D pictures and graphs. We demonstrate just a few of these capabilities here; see the Sage documentation for many more plotting capabilities.

Examples:

1. **Vectors**: `v.plot(start=u)` plots a vector **v** starting at **u**. Adding graphics objects superimposes them. Use `show()` to show a plot or use the `save()` method of a graphics object to save the plot to a file. We can save plots in a variety of formats (e.g., png, jpg, pdf, eps, svg, etc.). See Figure 91.1.

```
sage: u = vector(QQ, [1,2]); v = vector(QQ, [1,-1])
sage: p = v.plot() + u.plot(start=v) + (u+v).plot(linestyle='dashed')
sage: show(p) # show picture
sage: p.save('vectorplot.pdf') # save as a pdf file
```

2. **Matrices**: Sage plots matrices as rectangular arrays by mapping entry values to colors. A user-specified color map (rules for how values map to colors) can be given and a legend-like color bar can be added. In the following example (Figure 91.2a), we plot a Toeplitz matrix; you can clearly see the banded structure.

```
sage: from scipy.linalg import toeplitz
sage: A = matrix(QQ,toeplitz([cos(i) for i in range(10)]))
sage: A.plot(colorbar=True)
```

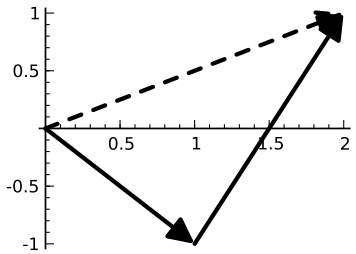
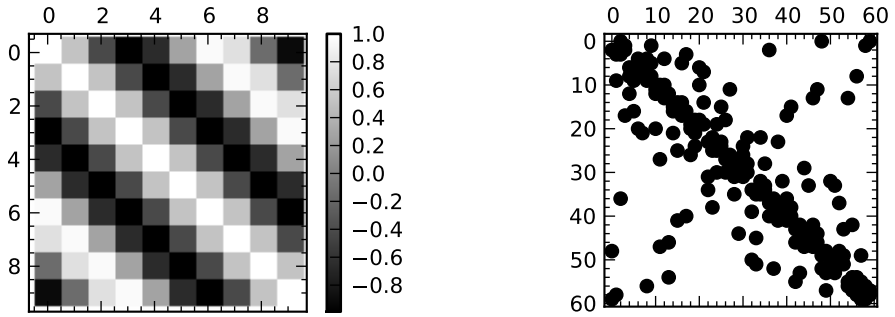


FIGURE 91.1: Vector plot.



(a) (b)

FIGURE 91.2: Matrix plots.

Sparse matrices plot their nonzero values. Here we construct the adjacency matrix of the Buckyball graph and plot its structure (Figure 91.2b).

```
sage: A = graphs.BuckyBall().adjacency_matrix(sparse=True)
sage: A.plot()
```

91.9 Conversion to Other Forms

Examples:

1. Sage can convert a vector or matrix to a Python dictionary (`v.dict()`; returns indices and entries), a Python list (`v.list()`), or a NumPy array (`v.numpy()`).

   ```
   sage: v = vector(QQ, [1,2,3]); v.dict()
   {0: 1, 1: 2, 2: 3}
   sage: A = matrix(QQ, [[1,2],[3,4]]); A.list()
   [1, 2, 3, 4]
   ```

2. The LaTeX code for a vector or matrix can be generated using the `latex()` method:

```
sage: latex(vector(QQ, [1,2,3]))
\left(1,\,2,\,3\right)
sage: latex(matrix(QQ, 2, [1,2,3,4/7]))
\left(\begin{array}{rr}
1 & 2 \\
3 & \frac{4}{7}
\end{array}\right)
```

3. You can change LaTeX delimiters using the following functions: `latex.vector_delimiters()`, `latex.matrix_delimiters()`. If you use a delimiter that has a backslash, use two backslashes when specifying it, like `'\\langle'`, or prefix the string with `r`, like `r'\langle'`.

```
sage: latex.vector_delimiters('\\langle', r'\rangle')
sage: latex(vector(QQ, [1,2,3/7]))
\left\langle1,\,2,\,\frac{3}{7}\right\rangle
```

91.10 General Rings

As mentioned earlier, every matrix and vector knows what ring in which its entries live, and this determines how a matrix prints, algorithms and libraries used to do the computations, methods available, etc. Sage also supports many more rings and fields than we have mentioned, including cyclotomic fields, number fields, and univariate and multivariate polynomial rings.

Examples:

1. The following matrix is invertible over the rationals, but not over the integers:

```
sage: A = matrix(QQ,2,[1,2,3,4])
sage: A.is_invertible(), A.change_ring(ZZ).is_invertible()
(True, False)
```

2. Sometimes matrices over certain fields support additional methods. For example, a symbolic matrix has a `subs()` method to substitute values in for variables:

```
sage: x,y,z = var('x,y,z'); A = matrix(SR, [[x,x-3*y,z],[x*z,0,y-x]])
sage: A.subs(x=3, y=2)
[  3  -3   z]
[3*z   0  -1]
```

3. **Finite fields**: See Chapter 31 for more about matrices over finite fields.
   (a) Use the `FiniteField` command or its shorthand `GF` ("Galois Field") to construct a finite field, specifying a name for the (multiplicative) generator if needed.

```
sage: FiniteField(37), GF(next_prime(10^5))
(Finite Field of size 37, Finite Field of size 100003)
sage: F = GF(3^2, 'a'); F.list()
[0, 2*a, a + 1, a + 2, 2, a, 2*a + 2, 2*a + 1, 1]
sage: F.<a> = GF(2^9); a^101
a^7 + a^6 + a^4 + a^3 + 1
```

   (b) Linear algebra can be done over these finite fields just like any other ring. Special implementations take advantage of different field characteristics to efficiently implement these. Linear algebra over `GF(2)` is particularly optimized.

```
sage: A = matrix(GF(5), 2, [1,2,3,14]); A.charpoly()
x^2 + 3
sage: A + A, A^-1, A^-1 * A
(
[2 4]  [3 1]  [1 0]
[1 3], [4 2], [0 1]
)
sage: B = matrix(GF(37), 3, [1,2,3,4,5,6,7,8,10]); b = vector([1,2,3])
sage: B * b, B \ b
((14, 32, 16), (12, 13, 0))
```

(c) Matrices over finite fields also support extra methods. Comparing a matrix to 1 compares it to the identity matrix.

```
sage: A = matrix(GF(37), 3, [1..8,10])
sage: A.multiplicative_order(), A^684 == 1
(684, True)
```

91.11 Numerical Linear Algebra

Facts:

1. The fields `RDF` and `CDF` are fast machine-level double-precision floating point reals and complexes (respectively). Many of the algorithms for matrices over `RDF` and `CDF` are provided by SciPy/NumPy and use industry-standard libraries like LAPACK.
2. The fields `RealField(prec)` and `ComplexField(prec)` provide reals and complexes to any fixed precision (`prec` is the number of bits used for the mantissa), e.g., `RealField(20)` has 20 bits of precision. In Sage, `RR` is `RealField(53)` and `CC` is `ComplexField(53)`. A floating point number is in `RR` or `CC` by default, so a matrix or vector with floating point entries is over `RR` or `CC` unless another ring is specified.
3. We emphasize that `RR` and `CC` are *not* exactly the same as `RDF` and `CDF` since `RR` and `CC` are not implemented with machine-level floating point numbers. The `RealField` and `ComplexField` fields (like `RR` and `CC`) are instead implemented with rigorous and portable semantics, and thus are usually slower. More importantly, algorithms for matrices over `RealField`, and `ComplexField` are not specialized for numerical work, so they may return misleading results because of numerical error.
4. Since the `v.n()` and `A.n()` numerical approximation methods return objects over `RR` or `CC`, if you are doing numerical computations, it is preferable to convert your objects to `RDF` or `CDF` using the `change_ring()` method, e.g., `v.change_ring(RDF)`.
5. `RDF` and `CDF` matrices used specialized algorithms for computing eigenvalues, eigenvectors, and popular matrix decompositions (see the relevant sections of Section 91.4 for specific methods). Eigenvectors (and eigenspaces) are reported with multiplicity one, for consistency with the format of results for exact matrices.
6. `RDF` and `CDF` matrices use specialized algorithms to compute properties like the condition number of a matrix (`A.condition()`), norms of vectors and matrices, matrix exponentials, and checks for properties like unitary or Hermitian matrices.

91.12 Applications of Linear Algebra

Sage can do many of the computations mentioned in the *Handbook of Linear Algebra*.

1. **Graphs** [Chapters 38-40]: Sage has a comprehensive graph theory and combinatorics library. See "Graph Theory" and "Combinatorics" in the Sage reference manual.
2. **Coding Theory** [Chapter 73]: Sage includes a number of codes and functionality related to coding theory. See the "Coding Theory" section of the reference manual.
3. **Linear Programming** [Chapters 68-69]: Sage comes with powerful linear programming, integer programming, and semidefinite programming tools, and also interfaces seamlessly with several industry solutions, such as CPLEX. See the "Numerical Tools" and "Numerical Optimization" sections of the reference manual.
4. **Minimum rank, zero forcing** [Chapter 46]: A Sage library that calculates minimum rank and zero forcing is at `https://github.com/jasongrout/minimum_rank`.

91.13 For More Information

1. **Web sites**: The main web site for Sage is `http://sagemath.org`. The main public Sage notebook server is `http://sagenb.org`.
2. **Sage help**: Access Sage documentation by clicking "Help" in the Sage Notebook or at `http://sagemath.org/doc`. Questions about how to use Sage should be addressed to either the sage-support mailing list, `http://groups.google.com/group/sage-support/`, or to the `http://ask.sagemath.org` web site. Discussion about Sage in education also happens on the sage-edu mailing list, `http://groups.google.com/group/sage-edu/`.
3. **Interacts**: Sage makes it very easy to create interactive demonstrations that utilize sliders, checkboxes, buttons, etc., allowing a user to easily adjust the inputs to a computation. See a number of examples at `http://wiki.sagemath.org/interact/` or the documentation for `interact()` (i.e., `interact?`) in the Sage notebook.
4. **Embedding Sage in a webpage**: Sage computations can easily be embedded in any webpage outside of the Sage notebook. A user can modify the Sage code, drag sliders, etc. See `https://sagecell.sagemath.org`.
5. **SageTeX**: Sage has a very easy way to embed Sage input and output directly into a LaTeX file. This allows an author to type Sage code directly into their LaTeX file, run the latex program (which generates a Sage code file), run Sage on the resulting Sage code file (which generates the outputs, graphics, etc.), and then run LaTeX again on their source file (which incorporates output and graphics back into the resulting pdf document). For example, to embed a Sage plot, merely type `\sageplot{plot(x^2, (x,0,4))}` in the LaTeX document, then run LaTeX, Sage, then LaTeX again to embed the plot in the pdf file. See the documentation for SageTeX in the Sage tutorial.
6. **Fortran support**: It is easy to compile and run FORTRAN code from within the Sage notebook. For examples, see *Numerical Sage* in the Sage documentation.

The following resources may also be helpful.

1. *Linear Algebra Quick Reference* by Robert Beezer. A two-page summary of many linear algebra commands and constructions. `http://wiki.sagemath.org/quickref`.
2. *Linear Algebra with Sage* by Robert Beezer. An extensive tutorial on linear algebra, designed as a supplement to the textbook *A First Course in Linear Algebra*. `http://linear.ups.edu/sage-fcla.html`.

3. Many numerical calculations and plots can be done using the included Python packages SciPy and NumPy (`http://www.scipy.org/`) and Matplotlib (`http://matplotlib.sourceforge.net/`).
4. *Python Scientific Lecture Notes*. `http://scipy-lectures.github.com/`.
5. *NumPy for Matlab Users*. Helps in translating between MATLAB and the Python NumPy package. `http://www.scipy.org/NumPy_for_Matlab_Users`. See also `http://mathesaurus.sourceforge.net/matlab-numpy.html` for another reference.
6. *Python documentation*. The official documentation is quite comprehensive and contains tutorials and reference material. `http://docs.python.org/`.
7. *Python for Non-Programmers*. A collection of resources introducing Python. `http://wiki.python.org/moin/BeginnersGuide/NonProgrammers`.
8. *Dive into Python* by Mark Pilgrim. A free comprehensive book introducing Python. `http://www.diveintopython.net/`.