Sage Primer for Linear Algebra

A First Course in Linear Algebra

Robert A. Beezer

University of Puget Sound

Version 3.30 (August 27, 2014)

Contents

Chapter SLE
Systems of Linear Equations

Section SSLE: Solving Systems of Linear Equations

GS Getting Started

Sage is a powerful system for studying and exploring many different areas of mathematics. In the next section, and the majority of the remaining section, we will include short descriptions and examples using Sage. You can read a bit more about Sage in the Preface. If you are not already reading this in an electronic version, you may want to investigate obtaining the worksheet version of this book, where the examples are "live" and editable. Most of your interaction with Sage will be by typing commands into a *compute cell*. That is a compute cell just below this paragraph. Click once inside the compute cell and you will get a more distinctive border around it, a blinking cursor inside, plus a cute little "evaluate" link below it.

```
```

At the cursor, type `2+2` and then click on the evaluate link. Did a `4` appear below the cell? If so, you have successfully sent a command off for Sage to evaluate and you have received back the (correct) answer.

Here is another compute cell. Try evaluating the command `factorial(300)`.

```
```

Hmmmmm. That is quite a big integer! The slashes you see at the end of each line mean the result is continued onto the next line, since there are 615 digits in the result.

To make new compute cells, hover your mouse just above another compute cell, or just below some output from a compute cell. When you see a skinny blue bar across the width of your worksheet, click and you will open up a new compute cell, ready for input. Note that your worksheet will remember any calculations you make, in the order you make them, no matter where you put the cells, so it is best to stay organized and add new cells at the bottom.

Try placing your cursor just below the monstrous value of 300! that you have. Click on the blue bar and try another factorial computation in the new compute cell.

Each compute cell will show output due to only the very last command in the cell. Try to predict the following output before evaluating the cell.

```
a = 10
b = 6
```

```
a = a + 20
a
```

```
30
```

The following compute cell will not print anything since the one command does not create output. But it will have an effect, as you can see when you execute the subsequent cell. Notice how this uses the value of `b` from above. Execute this compute cell *once*. Exactly once. Even if it *appears* to do nothing. If you execute the cell twice, your credit card may be charged twice.

```
b = b + 50
```

Now execute this cell, which will produce some output.

```
b + 20
```

```
76
```

So `b` came into existence as `6`. Then a cell added `50`. This assumes you only executed this cell once! In the last cell we create `b+20` (but do not save it) and it is this value that is output.

You can combine several commands on one line with a semi-colon. This is a great way to get multiple outputs from a compute cell. The syntax for building a matrix should be somewhat obvious when you see the output, but if not, it is not particularly important to understand now.

```
f(x) = x^8 - 7*x^4; f
```

```
x |--> x^8 - 7*x^4
```

```
f; print ; f.derivative()
```

```
x |--> x^8 - 7*x^4
<BLANKLINE>
x |--> 8*x^7 - 28*x^3
```

```
g = f.derivative()
g.factor()
```

```
4*(2*x^4 - 7)*x^3
```

Some commands in Sage are "functions," an example is `factorial()` above. Other commands are "methods" of an object and are like characteristics of objects, examples are `.factor()` and `.derivative()` as methods of a function. To comment on your work, you can open up a small word-processor. Hover your mouse until you get the skinny blue bar again, but now when you click, also hold the SHIFT key at the same time. Experiment with fonts, colors, bullet lists, etc and then click the "Save changes" button to exit. Double-click on your text if you need to go back and edit it later.

Open the word-processor again to create a new bit of text (maybe next to the empty compute cell just below). Type all of the following *exactly*, but do not include any backslashes that might precede the dollar signs in the print version:

```
Pythagorean Theorem: \$c^2=a^2+b^2\$
```

and save your changes. The symbols between the dollar signs are written according to the mathematical typesetting language known as TeX — cruise the internet to learn more about this very popular tool. (Well, it is extremely popular among mathematicians and physical scientists.)

```
```

Much of our interaction with sets will be through Sage lists. These are not really sets — they allow duplicates, and order matters. But they are so close to sets, and so easy and powerful to use that we will use them regularly. We will use a fun made-up list for practice, the quote marks mean the items are just text, with no special mathematical meaning. Execute these compute cells as we work through them.

```
zoo = ['snake', 'parrot', 'elephant', 'baboon', 'beetle']
zoo
```

```
['snake', 'parrot', 'elephant', 'baboon', 'beetle']
```

So the square brackets define the boundaries of our list, commas separate items, and we can give the list a name. To work with just one element of the list, we use the name and a pair of brackets with an index. Notice that lists have indices that *begin counting at zero*. This will seem odd at first and will seem very natural later.

```
zoo[2]
```

```
'elephant'
```

We can add a new creature to the zoo, it is joined up at the far right end.

```
zoo.append('ostrich'); zoo
```

```
['snake', 'parrot', 'elephant', 'baboon', 'beetle', 'ostrich']
```

We can remove a creature.

```
zoo.remove('parrot')
zoo
```

```
['snake', 'elephant', 'baboon', 'beetle', 'ostrich']
```

We can extract a sublist. Here we start with element 1 (the elephant) and go all the way up to, *but not including*, element 3 (the beetle). Again a bit odd, but it will feel natural later. For now, notice that we are extracting two elements of the lists, exactly $3 - 1 = 2$ elements.

```
mammals = zoo[1:3]
mammals
```

```
['elephant', 'baboon']
```

Often we will want to see if two lists are equal. To do that we will need to sort a list first. A function creates a new, sorted list, leaving the original alone. So we need to save the new one with a new name.

```
newzoo = sorted(zoo)
newzoo
```

```
['baboon', 'beetle', 'elephant', 'ostrich', 'snake']
```

```
zoo.sort()
zoo
```

```
['baboon', 'beetle', 'elephant', 'ostrich', 'snake']
```

Notice that if you run this last compute cell your zoo has changed and some commands above will not necessarily execute the same way. If you want to experiment, go all the way back to the first creation of the zoo and start executing cells again from there with a fresh zoo.

A construction called a "list comprehension" is especially powerful, especially since it almost exactly mirrors notation we use to describe sets. Suppose we want to form the plural of the names of the creatures in our zoo. We build a new list, based on all of the elements of our old list.

```
plurality_zoo = [animal+'s' for animal in zoo]
plurality_zoo
```

```
['baboons', 'beetles', 'elephants', 'ostrichs', 'snakes']
```

Almost like it says: we add an "s" to each animal name, for each animal in the zoo, and place them in a new list. Perfect. (Except for getting the plural of "ostrich" wrong.)

One final type of list, with numbers this time. The `range()` function will create lists of integers. In its simplest form an invocation like `range(12)` will create a list of 12 integers, *starting at zero* and working up to, *but not including*, 12. Does this sound familiar?

```
dozen = range(12); dozen
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

Here are two other forms, that you should be able to understand by studying the examples.

```
teens = range(13, 20); teens
```

```
[13, 14, 15, 16, 17, 18, 19]
```

```
decades = range(1900, 2000, 10); decades
```

```
[1900, 1910, 1920, 1930, 1940, 1950, 1960, 1970, 1980, 1990]
```

There is a "Save" button in the upper-right corner of your worksheet. This will save a current copy of your worksheet that you can retrieve from within your notebook again later, though you have to re-execute all the cells when you re-open the worksheet later.

There is also a "File" drop-down list, on the left, just above your very top compute cell (not be confused with your browser's File menu item!). You will see a choice here labeled "Save worksheet to a file..." When you do this, you are creating a copy of your worksheet in the "sws" format (short for "Sage WorkSheet"). You can email this file, or post it on a website, for other Sage users and they can use the "Upload" link on their main notebook page to incorporate a copy of your worksheet into their notebook.

There are other ways to share worksheets that you can experiment with, but this gives you one way to share any worksheet with anybody almost anywhere.

We have covered a lot here in this section, so come back later to pick up tidbits you might have missed. There are also many more features in the notebook that we have not covered.

Section RREF: Reduced Row-Echelon Form

M Matrices

Matrices are fundamental objects in linear algebra and in Sage, so there are a variety of ways to construct a matrix in Sage. Generally, you need to specify what types of entries the matrix contains (more on that in a minute), the number of rows and columns, and the entries themselves. First, let us dissect an example:

```
A = matrix(QQ, 2, 3, [[1, 2, 3], [4, 5, 6]])
A
```

```
[1 2 3]
[4 5 6]
```

`QQ` is the set of all rational numbers (fractions with an integer numerator and denominator), `2` is the number of rows, `3` is the number of columns. Sage understands a list of items as delimited by brackets (`[,]`) and the items in the list can again be lists themselves. So `[[1, 2, 3], [4, 5, 6]]` is a list of lists, and in this context the inner lists are rows of the matrix.

There are various shortcuts you can employ when creating a matrix. For example, Sage is able to infer the size of the matrix from the lists of entries.

```
B = matrix(QQ, [[1, 2, 3], [4, 5, 6]])
B
```

```
[1 2 3]
[4 5 6]
```

Or you can specify how many rows the matrix will have and provide one big grand list of entries, which will get chopped up, row by row, if you prefer.

```
C = matrix(QQ, 2, [1, 2, 3, 4, 5, 6])
C
```

```
[1 2 3]
[4 5 6]
```

It is possible to also skip specifying the type of numbers used for entries of a matrix, however this is fraught with peril, as Sage will make an informed guess about your intent. Is this what you want? Consider when you enter the single character "2" into a computer program like Sage. Is this the integer 2, the rational number $\frac{2}{1}$, the real number 2.00000, the complex number $2 + 0i$, or the polynomial $p(x) = 2$? In context, us humans can usually figure it out, but a literal-minded computer is not so smart. It happens that the operations we can perform, and how they behave, are influenced by the type of the entries in a matrix. So it is important to get this right and our advice is to be explicit and be in the habit of always specifying the type of the entries of a matrix you create.

Mathematical objects in Sage often come from sets of similar objects. This set is called the "parent" of the element. We can use this to learn how Sage deduces the type of entries in a matrix. Execute the following three compute cells in the Sage notebook, and notice how the three matrices are constructed to have entries from the integers, the rationals and the reals.

```
A = matrix(2, 3, [[1, 2, 3], [4, 5, 6]])
A.parent()
```

```
Full MatrixSpace of 2 by 3 dense matrices over Integer Ring
```

```
B = matrix(2, 3, [[1, 2/3, 3], [4, 5, 6]])
B.parent()
```

```
Full MatrixSpace of 2 by 3 dense matrices over Rational Field
```

```
C = matrix(2, 3, [[1, sin(2.2), 3], [4, 5, 6]])
C.parent()
```

```
Full MatrixSpace of 2 by 3 dense matrices over
Real Field with 53 bits of precision
```

Sage knows a wide variety of sets of numbers. These are known as "rings" or "fields" (see Section F), but we will call them "number systems" here. Examples include: `ZZ` is the integers, `QQ` is the rationals, `RR` is the real numbers with reasonable precision, and `CC` is the complex numbers with reasonable precision. We will present the theory of linear algebra over the complex numbers. However, in any computer system, there will always be complications surrounding the inability of digital arithmetic to accurately represent all complex numbers. In contrast, Sage can represent rational numbers exactly as the quotient of two (perhaps very large) integers. So our Sage examples will begin by using `QQ` as our number system and we can concentrate on understanding the key concepts.

Once we have constructed a matrix, we can learn a lot about it (such as its parent). Sage is largely object-oriented, which means many commands apply to an object by using the "dot" notation. `A.parent()` is an example of this syntax, while the constructor `matrix([[1, 2, 3], [4, 5, 6]])` is an exception. Here are a few examples, followed by some explanation:

```
A = matrix(QQ, 2, 3, [[1,2,3],[4,5,6]])
A.nrows(), A.ncols()
```

```
(2, 3)
```

```
A.base_ring()
```

```
Rational Field
```

```
A[1,1]
```

```
5
```

```
A[1,2]
```

```
6
```

The number of rows and the number of columns should be apparent, `.base_ring()` gives the number system for the entries, as included in the information provided by `.parent()`.

Computer scientists and computer languages prefer to begin counting from zero, while mathematicians and written mathematics prefer to begin counting at one. Sage and this text are no exception. It takes some getting used to, but the reasons for counting from zero in computer programs soon becomes very obvious. Counting from one in mathematics is historical, and unlikely to change anytime soon. So above, the two rows of `A` are numbered 0 and 1, while the columns are numbered 0, 1 and 2. So `A[1,2]` refers to the entry of `A` in the second row and the third column, i.e. 6.

There is much more to say about how Sage works with matrices, but this is already a lot to digest. Use the space below to create some matrices (different ways) and examine them and their properties (size, entries, number system, parent).

V Vectors

Vectors in Sage are built, manipulated and interrogated in much the same way as matrices (see Sage M). However as simple lists ("one-dimensional," not "two-dimensional" such as matrices that look more tabular) they are simpler to construct and manipulate. Sage will print a vector across the screen, even if we wish to interpret it as a column vector. It will be delimited by parentheses (`(,)`) which allows us to distinguish a vector from a matrix with just one row, if we look carefully. The number of "slots" in a vector is not referred to in Sage as rows or columns, but rather by "degree." Here are some examples (remember to start counting from zero):

```
v = vector(QQ, 4, [1, 1/2, 1/3, 1/4])
v
```

```
(1, 1/2, 1/3, 1/4)
```

```
v.degree()
```

```
4
```

```
v.parent()
```

```
Vector space of dimension 4 over Rational Field
```

```
v[2]
```

```
1/3
```

```
w = vector([1, 2, 3, 4, 5, 6])
w
```

```
(1, 2, 3, 4, 5, 6)
```

```
w.degree()
```

```
6
```

```
w.parent()
```

```
Ambient free module of rank 6 over
the principal ideal domain Integer Ring
```

```
w[3]
```

```
4
```

Notice that if you use commands like `.parent()` you will sometimes see references to "free modules." This is a technical generalization of the notion of a vector, which is beyond the scope of this course, so just mentally convert to vectors when you see this term.

The zero vector is super easy to build, but be sure to specify what number system your zero is from.

```
z = zero_vector(QQ, 5)
z
```

```
(0, 0, 0, 0, 0)
```

Notice that while using Sage, we try to remain consistent with our mathematical notation conventions. This is not required, as you can give items in Sage very long names if you wish. For example, the following is perfectly legitimate, as you can see.

```
blatzo = matrix(QQ, 2, [1, 2, 3, 4])
blatzo
```

```
[1 2]
[3 4]
```

In fact, our use of capital letters for matrices actually contradicts some of the conventions for naming objects in Sage, so there are good reasons for *not* mirroring our mathematical notation.

AM Augmented Matrix

Sage has a matrix method, `.augment()`, that will join two matrices, side-by-side provided they both have the same number of rows. The same method will allow you to augment a matrix with a column vector, as described in Definition AM, provided the number of entries in the vector matches the number of rows for the matrix. Here we reprise the construction in Example AMAA. We will now format our matrices as input across several lines, a practice you may use in your own worksheets, or not.

```
A = matrix(QQ, 3, 3, [[1, -1, 2],
                      [2,  1, 1],
                      [1,  1, 0]])
b = vector(QQ, [1, 8, 5])
M = A.augment(b)
M
```

```
[ 1 -1  2  1]
[ 2  1  1  8]
[ 1  1  0  5]
```

Notice that the matrix method `.augment()` needs some input, in the above case, the vector `b`. This will explain the need for the parentheses on the end of the "dot" commands, even if the particular command does not expect input.

Some methods allow optional input, typically using keywords. Matrices can track subdivisions, making breaks between rows and/or columns. When augmenting, you can ask for the subdivision to be included. Evalute the compute cell above if you have not already, so that `A` and `b` are defined, and then evaluate:

```
M = A.augment(b, subdivide=True)
M
```

```
[ 1 -1  2| 1]
[ 2  1  1| 8]
[ 1  1  0| 5]
```

As a partial demonstration of manipulating subdivisions of matrices we can reset the subdivisions of `M` with the `.subdivide()` method. We provide a list of rows to subdivide *before*, then a list of columns to subdivide *before*, where we remember that counting begins at zero.

```
M.subdivide([1,2],[1])
M
```

```
[ 1|-1  2  1]
[--+--------]
[ 2| 1  1  8]
[--+--------]
[ 1| 1  0  5]
```

RO Row Operations

Sage will perform individual row operations on a matrix. This can get a bit tedious, but it is better than doing the computations (wrong, perhaps) by hand, and it can be useful when building up more complicated procedures for a matrix.

For each row operation, there are two similar methods. One changes the matrix "in-place" while the other creates a new matrix that is a modified version of the original. This is an important distinction that you should understand for every new Sage command you learn that might change a matrix or vector.

Consider the first row operation, which swaps two rows. There are two matrix methods to do this, a "with" version that will create a new, changed matrix, which you will likely want to save, and a plain version that will change the matrix it operates on "in-place.". The `copy()` function, which is a general-purpose command, is a way to make a copy of a matrix before you make changes to it. Study the example below carefully, and then read the explanation following. (Remember that counting begins with zero.)

```
A = matrix(QQ,2,3,[1,2,3,4,5,6])
B = A
C = copy(A)
D = A.with_swapped_rows(0,1)
D[0,0] = -1
A.swap_rows(0,1)
A[1,2] = -6
A
```

```
[ 4  5  6]
[ 1  2 -6]
```

```
B
```

```
[ 4  5  6]
[ 1  2 -6]
```

```
C
```

```
[1 2 3]
[4 5 6]
```

```
D
```

```
[-1  5  6]
[ 1  2  3]
```

Here is how each of these four matrices comes into existence.

1. `A` is our original matrix, created from a list of entries.
2. `B` is not a new copy of `A`, it is just a new name for referencing the exact same matrix internally.
3. `C` is a brand new matrix, stored internally separate from `A`, but with identical contents.
4. `D` is also a new matrix, which is created by swapping the rows of `A`

And here is how each matrix is affected by the commands.

1. `A` is changed twice "in-place". First, its rows are swapped rows a "plain" matrix method. Then its entry in the lower-right corner is set to `-6`.
2. `B` is just another name for `A`. So whatever changes are made to `A` will be evident when we ask for the matrix by the name `B`. And vice-versa.
3. `C` is a copy of the original `A` and does not change, since no subsequent commands act on `C`.
4. `D` is a new copy of `A`, created by swapping the rows of `A`. Once created from `A`, it has a life of its own, as illustrated by the change in its entry in the upper-left corner to `-1`.

An interesting experiment is to rearrange some of the lines above (or add new ones) and predict the result.

Just as with the first row operation, swapping rows, Sage supports the other two row operations with natural sounding commands, with both "in-place" versions and new-matrix versions.

```
A = matrix(QQ, 3, 4, [[1,  2,  3,  4],
                      [5,  6,  7,  8],
                      [9, 10, 11, 12]])
B = copy(A)
A.rescale_row(1, 1/2)
A
```

```
[  1   2   3   4]
[5/2   3 7/2   4]
[  9  10  11  12]
```

```
A.add_multiple_of_row(2, 0, 10)
A
```

```
[  1   2   3   4]
[5/2   3 7/2   4]
[ 19  30  41  52]
```

```
B.with_rescaled_row(1, 1/2)
```

```
[  1   2   3   4]
[5/2   3 7/2   4]
[  9  10  11  12]
```

```
C = B.with_added_multiple_of_row(2, 0, 10)
C
```

```
[ 1  2  3  4]
[ 5  6  7  8]
[19 30 41 52]
```

Notice that the order of the arguments might feel a bit odd, compared to how we write and think about row operations. Also note how the "with" versions leave a trail of matrices for each step while the plain versions just keep changing `A`.

RREF Reduced Row-Echelon Form

There has been a lot of information about using Sage with vectors and matrices in this section. But we can now construct the coefficient matrix of a system of equations and the vector of constants. From these pieces we can easily construct the augmented matrix, which we could subject to a series of row operations. Computers are suppose to make routine tasks easy so we can concentrate on bigger ideas. No exception here, Sage can bring a matrix (augmented or not) to reduced row echelon form with no problem. Let us redo Example SAB with Sage.

```
coeff = matrix(QQ, [[-7, -6, -12],
                    [5,   5,   7],
                    [1,   0,   4]])
const = vector(QQ, [-33, 24, 5])
aug = coeff.augment(const)
aug.rref()
```

```
[ 1  0  0 -3]
[ 0  1  0  5]
[ 0  0  1  2]
```

And the solution $x_1 = -3$, $x_2 = 5$, $x_3 = 2$ is now obvious. Beautiful.

You may notice that Sage has some commands with the word "echelon" in them. For now, these should be avoided like the plague, as there are some subtleties in how they work. The matrix method `.rref()` will be sufficient for our purposes for a long, long time — so stick with it.

Section TSS: Types of Solution Sets

FDV Free and Dependent Variables

Sage has the matrix method `.pivot()` to quickly and easily identify the pivot columns of the reduced row-echelon form of a matrix. Notice that we do not have to row-reduce the matrix first, we just ask which columns of a matrix A *would be* the pivot columns of the matrix B that is row-equivalent to A and in reduced row-echelon form. By Definition IDV, the indices of the pivot columns for an augmented matrix of a system of equations are the indices of the dependent variables. And the remainder are free variables. But be careful, Sage numbers columns starting from zero and mathematicians typically number variables starting from one.

Let us reprise Example ISSI.

```
coeff = matrix(QQ, [[ 1,  4, 0, -1,  0,   7, -9],
                    [ 2,  8,-1,  3,  9, -13,  7],
                    [ 0,  0, 2, -3, -4,  12, -8],
                    [-1, -4, 2,  4,  8, -31, 37]])
```

```
const = vector(QQ, [3, 9, 1, 4])
aug = coeff.augment(const)
dependent = aug.pivots()
dependent
```

```
(0, 2, 3)
```

So, incrementing each column index by 1 gives us the same set D of indices for the dependent variables. To get the free variables, we can use the following code. Study it and then read the explanation following.

```
free = [index for index in range(7) if not index in dependent]
free
```

```
[1, 4, 5, 6]
```

This is a Python programming construction known as a "list comprehension" but in this setting I prefer to call it "set builder notation." Let us dissect the command in pieces. The brackets (`[,]`) create a new list. The items in the list will be values of the variable `index`. `range(7)` is another list, integers starting at `0` and stopping *just before* `7`. (While perhaps a bit odd, this works very well when we consistently start counting at zero.) So `range(7)` is the list `[0,1,2,3,4,5,6]`. Think of these as candidate values for `index`, which are generated by `for index in range(7)`. Then we test each candidate, and keep it in the new list if it is *not* in the list `dependent`.

This is entirely analogous to the following mathematics:

$$F = \{\, f \mid 1 \leq f \leq 7, f \notin D\}$$

where F is `free`, f is `index`, and D is `dependent`, and we make the 0/1 counting adjustments. This ability to construct sets in Sage with notation so closely mirroring the mathematics is a powerful feature worth mastering. We will use it repeatedly. It was a good exercise to use a list comprehension to form the list of columns that are not pivot columns. However, Sage has us covered.

```
free_and_easy = coeff.nonpivots()
free_and_easy
```

```
(1, 4, 5, 6)
```

Can you use this new matrix method to make a simpler version of the `consistent()` function we designed above?

RCLS Recognizing Consistency of a Linear System

Another way of expressing Theorem RCLS is to say a system is consistent if and only if column $n + 1$ is not a pivot column of B. Sage has the matrix method `.pivot()` to easily identify the pivot columns of a matrix. Let us use Archetype E as an example.

```
coeff = matrix(QQ, [[ 2, 1,  7, -7],
                    [-3, 4, -5, -6],
                    [ 1, 1,  4, -5]])
const = vector(QQ, [2, 3, 2])
aug = coeff.augment(const)
```

```
aug.rref()
```

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

```
aug.pivots()
```

```
(0, 1, 4)
```

We can *look* at the reduced row-echelon form of the augmented matrix and see a pivot column in the final column, so we know the system is inconsistent. However, we could just as easily not form the reduced row-echelon form and just look at the list of pivot columns computed by `aug.pivots()`. Since `aug` has 5 columns, the final column is numbered `4`, which is present in the list of pivot columns, as we expect.

One feature of Sage is that we can easily extend its capabilities by defining new commands. Here will create a function that checks if an augmented matrix represents a consistent system of equations. The syntax is just a bit complicated. `lambda` is the word that indicates we are making a new function, the input is temporarily named `A` (think `A`ugmented), and the *name* of the function is `consistent`. Everything following the colon will be evaluated and reported back as the output.

```
consistent = lambda A: not(A.ncols()-1 in A.pivots())
```

Execute this block above. There will not be any output, but now the `consistent` function will be defined and available. Now give it a try (after making sure to have run the code above that defines `aug`). Note that the output of `consistent()` will be either `True` or `False`.

```
consistent(aug)
```

```
False
```

The `consistent()` command works by simply checking to see if the last column of `A` is *not* in the list of pivots. We can now test many different augmented matrices, such as perhaps changing the vector of constants while keeping the coefficient matrix fixed. Again, make sure you execute the code above that defines `coeff` and `const`.

```
consistent(coeff.augment(const))
```

```
False
```

```
w = vector(QQ, [3,1,2])
consistent(coeff.augment(w))
```

```
True
```

```
u = vector(QQ, [1,3,1])
consistent(coeff.augment(u))
```

```
False
```

Why do some vectors of constants lead to a consistent system with this coefficient matrix, while others do not? This is a fundamental question, which we will come to understand in several different ways.

SS1 Solving Systems, Part 1

Sage has a built-in command that will solve a linear system of equations, given a coefficient matrix and a vector of constants. We need to learn some more theory before we can entirely understand this command, but we can begin to explore its use. For now, consider these methods experimental and do not let it replace row-reducing augmented matrices.

The matrix method `A.solve_right(b)` will provide information about solutions to the linear system of equations with coefficient matrix `A` and vector of constants `b`. The reason for the "right" (and the corresponding command named with "left") will have to wait for Sage MVP. For now, it is generally correct in this course to use the "right" variant of any Sage linear algebra command that has both "left" and "right" variants.

Let us apply the `.solve_right()` command to a system with no solutions, in particular Archetype E. We have already seen in Sage RCLS that this system is inconsistent.

```
coeff = matrix(QQ, [[ 2, 1,  7, -7],
                    [-3, 4, -5, -6],
                    [ 1, 1,  4, -5]])
const = vector(QQ, [2, 3, 2])
coeff.solve_right(const)
```

```
Traceback (most recent call last):
...
ValueError: matrix equation has no solutions
```

This is our first discussion of Sage error messages, though undoubtedly you have seen several already! First, here we only show the first and last lines of the message since typically it contains a lot of information specific to whichever computer you may be using. but we always begin with the last line as the most important indication of what has happened. Here the "problem" is quite evident: we get an "error" message telling us that the `matrix equation has no solutions`. We can debate whether or not this is really an error, but that is the design decision taken in Sage — we just need to be aware of it, the `.solve_right()` is really only valuable when there is a solution.

Generally, when deciphering Sage error messages, you want to start at the bottom of the "traceback" and read up through the various routines that have been called. Execute the block above and you may see references to matrix methods such as `._solve_right_general()` and then `.solve_right()`. With time and practice, these mysterious messages will become more and more helpful, so spend some time reading them in tandem with locating the real source of any problems you encounter.

What does `.solve_right()` do with a system that does have solutions? Let us take a look at Example ISSI again, as we did in Sage FDV.

```
coeff = matrix(QQ, [[ 1,  4,  0, -1,  0,   7, -9],
                    [ 2,  8, -1,  3,  9, -13,  7],
                    [ 0,  0,  2, -3, -4,  12, -8],
```

```
                    [-1, -4,  2,  4,  8, -31, 37]])
const = vector(QQ, [3, 9, 1, 4])
coeff.solve_right(const)
```

```
(4, 0, 2, 1, 0, 0, 0)
```

This vector with 7 entries is indeed a solution to the system of equations (check this!). But from Example ISSI we know this system has *infinitely* many solutions. Why does Sage give us just one solution? Of the infinitely many solutions, why this one? How can a finite computer *ever* present us with infinitely many solutions? Do we have the time to read through an infinite list of solutions? Is there a "best" solution? This behavior should prompt these questions, and maybe more.

In order to totally understand the behavior of the `.solve_right()` command, we need to understand more of the theory of linear algebra. In good time. So for now, `.solve_right()` is a curiosity we will fully understand soon — specifically in Sage SS2 and Sage SS3.

Section HSE: Homogeneous Systems of Equations

SHS Solving Homogeneous Systems

We can explore homogeneous systems easily in Sage with commands we have already learned. Notably, the `zero_vector()` constructor will quickly create the necessary vector of constants (Sage V).

You could try defining a function to accept a matrix as input, augment the matrix with a zero vector having the right number of entries, and then return the reduced row-echelon form of the augmented matrix, or maybe you could return the number of free variables. (See Sage RCLS for a refresher on how to do this). It is also interesting to see how `.solve_right()` behaves on homogeneous systems, since in particular we know it will never give us an error. (Why not? Hint: Theorem HSC.)

NS Null Space

Sage will compute a null space for us. Which is rather remarkable, as it is an infinite set! Again, this is a powerful command, and there is lots of associated theory, so we will not understand everything about it right away, and it also has a radically different name in Sage. But we will find it useful immediately. Let us reprise Example NSEAI. The relevant command to build the null space of a matrix is `.right_kernel()`, where again, we will rely exclusively on the "right" version. Also, to match our work in the text, and make the results more recognizable, we will consistently us the keyword option `basis='pivot'`, which we will be able to explain once we have more theory (Sage SSNS, Sage SUTH0). Note too, that this is a place where it is critical that matrices are defined to use the rationals as their number system (`QQ`).

```
I = matrix(QQ, [[ 1,  4,  0, -1,  0,   7, -9],
                [ 2,  8, -1,  3,  9, -13,  7],
                [ 0,  0,  2, -3, -4,  12, -8],
                [-1, -4,  2,  4,  8, -31, 37]])
nsp = I.right_kernel(basis='pivot')
nsp
```

```
Vector space of degree 7 and dimension 4 over Rational Field
User basis matrix:
[-4  1  0  0  0  0  0]
```

```
[-2  0 -1 -2  1  0  0]
[-1  0  3  6  0  1  0]
[ 3  0 -5 -6  0  0  1]
```

As we said, `nsp` contains a lot of unfamiliar information. Ignore most of it for now. But as a set, we can test membership in `nsp`.

```
x = vector(QQ, [3, 0, -5, -6, 0, 0, 1])
x in nsp
```

```
True
```

```
y = vector(QQ, [-4, 1, -3, -2, 1, 1, 1])
y in nsp
```

```
True
```

```
z = vector(QQ, [1, 0, 0, 0, 0, 0, 2])
z in nsp
```

```
False
```

We did a bad thing above, as Sage likes to use `I` for the imaginary number $i = \sqrt{-1}$ and we just clobbered that. We will not do it again. See below how to fix this. `nsp` is an infinite set. Since we know the null space is defined as solution to a system of equations, and the work above shows it has at least two elements, we are not surprised to discover that the set is infinite (Theorem PSSLS).

```
nsp.is_finite()
```

```
False
```

If we want an element of the null space to experiment with, we can get a "random" element easily. Evaluate the following compute cell repeatedly to get a feel for the variety of the different output. You will see a different result each time, and the result supplied in your downloaded worksheet is very unlikely to be a result you will ever see again. The bit of text, `# random`, is technically a "comment", but we are using it as a signal to our automatic testing of the Sage examples that this example should be skipped. You *do not* need to use this device in your own work, though you may use the comment syntax if you wish.

```
z = nsp.random_element()
z                       # random
```

```
(21/5, 1, -102/5, -204/5, -3/5, -7, 0)
```

```
z in nsp
```

```
True
```

Sometimes, just sometimes, the null space is finite, and we can list its elements. This is from Example CNS2.

```
C = matrix(QQ, [[-4, 6, 1],
                [-1, 4, 1],
                [ 5, 6, 7],
                [ 4, 7, 1]])
Cnsp = C.right_kernel(basis='pivot')
Cnsp.is_finite()
```

```
True
```

```
Cnsp.list()
```

```
[(0, 0, 0)]
```

Notice that we get back a list (which mathematically is really a set), and it has one element, the three-entry zero vector.

There is more to learn about exploring the null space with Sage's `.right_kernel()` so we will see more of this matrix method. In the meantime, if you are done experimenting with the matrix `I` we can restore the variable `I` back to being the imaginary number $i = \sqrt{-1}$ with the Sage `restore()` command.

```
restore()
I^2
```

```
-1
```

SH Sage Help

There are many ways to learn about, or remind yourself of, how various Sage commands behave. Now that we have learned a few, it is a good time to show you the most direct methods of obtaining help. These work throughout Sage, so can be useful if you want to apply Sage to other areas of mathematics.

The first hurdle is to learn how to make a mathematical object in Sage. We know now how to make matrices and vectors (and null spaces). This is enough to help us explore relevant commands in Sage for linear algebra. First, define a very simple matrix `A`, with maybe one with one row and two columns. The number system you choose will have some effect on the results, so use `QQ` for now. In the notebook, enter `A.` (assuming you called your matrix `A`, and be sure to include the period). Now hit the "tab" key and you will get a long list of all the possible methods you can apply to `A` using the dot notation.

You can click directly on one of these commands (the word, not the blue highlight) to enter it into the cell. Now instead of adding parentheses to the command, place a single question mark (`?`) on the end and hit the tab key again. You should get some nicely formatted documentation, along with example uses. (Try `A.rref?` below for a good example of this.) You can replace the single question mark by two question marks, and as Sage is an open source program you can see the actual computer instructions for the method, which at first includes all the documentation again. Note that now the documentation is enclosed in a pair of triple quotation marks (`"""`,`"""`) as part of the source code, and is not specially formatted.

These methods of learning about Sage are generally referred to as "tab-completion" and we will use this term going forward. To learn about the use of Sage in other areas of mathematics, you just need to find out how to create the relevant objects via a "constructor" function, such as `matrix()` and `vector()` for linear algebra.

Sage has a comprehensive Reference Manual and there is a Linear Algebra Quick Reference sheet. These should be easily located online via sagemath.org or with an internet search leading with the terms "sage math" (use "math" to avoid confusion with other websites for things named "Sage").

Section NM: Nonsingular Matrices

NM Nonsingular Matrix

Being nonsingular is an important matrix property, and in such cases Sage contains commands that quickly and easily determine if the mathematical object does, or does not, have the property. The names of these types of methods universally begin with `.is_`, and these might be referred to as "predicates" or "queries.". In the Sage notebook, define a simple matrix `A`, and then in a cell type `A.is_`, followed by pressing the tab key rather than evaluating the cell. You will get a list of numerous properties that you can investigate for the matrix `A`. (This will not work as advertised with the Sage cell server.) The other convention is to name these properties in a positive way, so the relevant command for nonsingular matrices is `.is_singular()`. We will redo Example S and Example NM. Note the use of `not` in the last compute cell.

```
A = matrix(QQ, [[1, -1, 2],
                [2,  1, 1],
                [1,  1, 0]])
A.is_singular()
```

```
True
```

```
B = matrix(QQ, [[-7, -6, -12],
                [ 5,  5,   7],
                [ 1,  0,   4]])
B.is_singular()
```

```
False
```

```
not(B.is_singular())
```

```
True
```

IM Identity Matrix

It is straightforward to create an identity matrix in Sage. Just specify the number system and the number of rows (which will equal the number of columns, so you do not specify that since it would be redundant). The number system can be left out, but the result will have entries from the integers (`ZZ`), which in this course is unlikely to be what you really want.

```
id5 = identity_matrix(QQ, 5)
id5
```

```
[1 0 0 0 0]
[0 1 0 0 0]
[0 0 1 0 0]
```

```
[0 0 0 1 0]
[0 0 0 0 1]
```

```
id4 = identity_matrix(4)
id4.base_ring()
```

```
Integer Ring
```

Notice that we do not use the now-familiar dot notation to *create* an identity matrix. What would we use the dot notation on anyway? For these reasons we call the `identity_matrix()` function a **constructor**, since it builds something from scratch, in this case a very particular type of matrix. We mentioned above that an identity matrix is in reduced row-echelon form. What happens if we try to row-reduce a matrix that is already in reduced row-echelon form? By the uniqueness of the result, there should be no change. The following code illustrates this. Notice that `=` is used to *assign* an object to a new name, while `==` is used to *test equality* of two objects. I frequently make the mistake of forgetting the second equal sign when I mean to test equality.

```
id50 = identity_matrix(QQ, 50)
id50 == id50.rref()
```

```
True
```

NME1 Nonsingular Matrix Equivalences, Round 1

Sage will create random matrices and vectors, sometimes with various properties. These can be very useful for quick experiments, and they are also useful for *illustrating* that theorems hold for any object satisfying the hypotheses of the theorem. But this will never replace a proof.

We will illustrate Theorem NME1 using Sage. We will use a variant of the `random_matrix()` constructor that uses the `algorithm='unimodular'` keyword. We will have to wait for Chapter D before we can give a full explanation, but for now, understand that this command will *always* create a square matrix that is nonsingular. Also realize that there are square nonsingular matrices which will never be the output of this command. In other words, this command creates elements of just a subset of all possible nonsingular matrices.

So we are using random matrices below to illustrate properties predicted by Theorem NME1. Execute the first command to create a random nonsingular matrix, and notice that we only have to mark the output of `A` as random for our automated testing process. After a few runs, notice that you can also edit the value of `n` to create matrices of different sizes. With a matrix `A` defined, run the next three cells, which by Theorem NME1 each always produce `True` as their output, *no matter what value* `A` *has*, so long as `A` is nonsingular. Read the code and try to determine exactly how they correspond to the parts of the theorem (some commentary along these lines follows).

```
n = 6
A = random_matrix(QQ, n, algorithm='unimodular')
A               # random
```

```
[   1   -4    8   14    8   55]
[   4  -15   29   50   30  203]
[  -4   17  -34  -59  -35 -235]
[  -1    3   -8  -16   -5  -48]
```

```
[  -5   16  -33  -66  -16 -195]
[   1   -2    2    7   -2   10]
```

```
A.rref() == identity_matrix(QQ, n)
```

```
True
```

```
nsp = A.right_kernel(basis='pivot')
nsp.list() == [zero_vector(QQ, n)]
```

```
True
```

```
b = random_vector(QQ, n)
aug = A.augment(b)
aug.pivots() == tuple(range(n))
```

```
True
```

The only portion of these commands that may be unfamilar is the last one. The command `range(n)` is incredibly useful, as it will create a list of the integers from `0` up to, *but not including*, `n`. (We saw this command briefly in Sage FDV.) So, for example, `range(3) == [0,1,2]` is `True`. Pivots are returned as a "tuple" which is very much like a list, except we cannot change the contents. We can see the difference by the way the tuple prints with parentheses (`(,)`) rather than brackets (`[,]`). We can convert a list to a tuple with the `tuple()` command, in order to make the comparison succeed.

How do we tell if the reduced row-echelon form of the augmented matrix of a system of equations represents a system with a unique solution? First, the system must be consistent, which by Theorem RCLS means the last column is not a pivot column. Then with a consistent system we need to insure there are no free variables. This happens if and only if the remaining columns are all pivot columns, according to Theorem FVCS, thus the test used in the last compute cell.

Chapter V
Vectors

Section VO: Vector Operations

VSCV Vector Spaces of Column Vectors

It is possible to construct vector spaces several ways in Sage. For now, we will show you two basic ways. Remember that while our theory is all developed over the complex numbers, $\mathbb{C}$, it is better to initially illustrate these ideas in Sage using the rationals, `QQ`.

To create a vector space, we use the `VectorSpace()` constructor, which requires the name of the number system for the entries and the number of entries in each vector. We can display some information about the vector space, and with tab-completion you can see what functions are available. We will not do too much with these methods immediately, but instead learn about them as we progress through the theory.

```
V = VectorSpace(QQ, 8)
V
```

```
Vector space of dimension 8 over Rational Field
```

Notice that the word "dimension" is used to refer to the number of entries in a vector contained in the vector space, whereas we have used the word "degree" before. Try pressing the Tab key while in the next cell to see the range of methods you can use on a vector space.

```
V.
```

We can easily create "random" elements of any vector space, much as we did earlier for the kernel of a matrix. Try executing the next compute cell several times.

```
w = V.random_element()
w             # random
```

```
(2, -1/9, 0, 2, 2/3, 0, -1/3, 1)
```

Vector spaces are a fundamental objects in Sage and in mathematics, and Sage has a nice compact way to create them, mimicking the notation we use when working on paper.

```
U = CC^5
U
```

```
Vector space of dimension 5 over
Complex Field with 53 bits of precision
```

```
W = QQ^3
W
```

```
Vector space of dimension 3 over Rational Field
```

Sage can determine if two vector spaces are the same. Notice that we use two equals sign to test equality, since we use a single equals sign to make assignments.

```
X = VectorSpace(QQ, 3)
W = QQ^3
X == W
```

```
True
```

VO Vector Operations

Sage can easily perform the two basic operations with vectors, vector addition, `+`, and scalar vector multiplication, `*`. Notice that Sage is not confused by an ambiguity due to multiple meanings for the symbols `+` and `*` — for example, Sage knows that `3 + 12` is different than the vector additions below.

```
x = vector(QQ, [1, 2, 3])
y = vector(QQ, [10, 20, 30])
5*x
```

```
(5, 10, 15)
```

```
x + y
```

```
(11, 22, 33)
```

```
3*x + 4*y
```

```
(43, 86, 129)
```

```
-y
```

```
(-10, -20, -30)
```

```
w = (-4/3)*x - (1/10)*y
w
```

```
(-7/3, -14/3, -7)
```

ANC A Note on Coercion

Study the following sequence of commands, while cognizant of the failure to specify a number system for `x`.

```
x = vector([1, 2, 3])
u = 3*x
u
```

```
(3, 6, 9)
```

```
v = (1/3)*x
v
```

```
(1/3, 2/3, 1)
```

```
y = vector(QQ, [4, 5, 6])
w = 8*y
w
```

```
(32, 40, 48)
```

```
z = x + y
z
```

```
(5, 7, 9)
```

None of this should be too much of a surprise, and the results should be what we would have expected. Though for `x` we never specified if `1, 2, 3` are integers, rationals, reals, complexes, or ...? Let us dig a little deeper and examine the parents of the five vectors involved.

```
x.parent()
```

```
Ambient free module of rank 3 over
the principal ideal domain Integer Ring
```

```
u.parent()
```

```
Ambient free module of rank 3 over
the principal ideal domain Integer Ring
```

```
v.parent()
```

```
Vector space of dimension 3 over Rational Field
```

```
y.parent()
```

```
Vector space of dimension 3 over Rational Field
```

```
w.parent()
```

```
Vector space of dimension 3 over Rational Field
```

```
z.parent()
```

```
Vector space of dimension 3 over Rational Field
```

So `x` and `u` belong to something called an "ambient free module," whatever that is. What is important here is that the parent of `x` uses the integers as its number system. How about `u`, `v`, `y`, `w`, `z`? All but the first has a parent that uses the rationals for its number system.

Three of the final four vectors are examples of a process that Sage calls "coercion." Mathematical elements get converted to a new parent, as necessary, when the conversion is totally unambiguous. In the examples above:

- `u` is the result of scalar multiplication by an integer, so the computation and result can all be accommodated within the integers as the number system.
- `v` involves scalar multiplication by a scalar that is not an integer, and which could be construed as a rational number. So the result needs to have a parent whose number system is the rationals.
- `y` is created *explicitly* as a vector whose entries are rational numbers.
- Even though `w` is created only with products of integers, the fact that `y` has entries considered as rational numbers, so too does the result.
- The creation of `z` is the result of adding a vector of integers to a vector of rationals. This is the best example of coercion — Sage promotes `x` to a vector of rationals and therefore returns a result that is a vector of rationals. Notice that there is no ambiguity and no argument about how to promote `x`, and the same would be true for any vector full of integers.

The coercion above is automatic, but we can also usually force it to happen without employing an operation.

```
t = vector([10, 20, 30])
t.parent()
```

```
Ambient free module of rank 3 over
the principal ideal domain Integer Ring
```

```
V = QQ^3
t_rational = V(t)
t_rational
```

```
(10, 20, 30)
```

```
t_rational.parent()
```

```
Vector space of dimension 3 over Rational Field
```

```
W = CC^3
t_complex = W(t)
t_complex
```

```
(10.0000000000000, 20.0000000000000, 30.0000000000000)
```

```
t_complex.parent()
```

```
Vector space of dimension 3 over
Complex Field with 53 bits of precision
```

So the syntax is to use the name of the parent like a function and *coerce* the element into the new parent. This can fail if there is no natural way to make the conversion.

```
u = vector(CC, [5*I, 4-I])
u
```

```
(5.00000000000000*I, 4.00000000000000 - 1.00000000000000*I)
```

```
V = QQ^2
V(u)
```

```
Traceback (most recent call last):
...
TypeError: Unable to coerce 5.00000000000000*I
(<type 'sage.rings.complex_number.ComplexNumber'>) to Rational
```

Coercion is one of the more mysterious aspects of Sage, and the above discussion may not be very clear the first time though. But if you get an error (like the one above) talking about coercion, you know to come back here and have another read through. For now, be sure to create all your vectors and matrices over `QQ` and you should not have any difficulties.

Section LC: Linear Combinations

LC Linear Combinations

We can redo Example TLC with Sage. First we build the relevant vectors and then do the computation.

```
u1 = vector(QQ,  [ 2, 4, -3,  1,  2, 9])
u2 = vector(QQ,  [ 6, 3,  0, -2,  1, 4])
u3 = vector(QQ,  [-5, 2,  1,  1, -3, 0])
u4 = vector(QQ,  [ 3, 2, -5,  7,  1, 3])
1*u1 + (-4)*u2 + 2*u3 +(-1)*u4
```

```
(-35, -6, 4, 4, -9, -10)
```

With a linear combination combining many vectors, we sometimes will use more compact ways of forming a linear combination. So we will redo the second linear combination of $\mathbf{u}_1$, $\mathbf{u}_2$, $\mathbf{u}_3$, $\mathbf{u}_4$ using a list comprehension and the `sum()` function.

```
vectors = [u1, u2, u3, u4]
scalars = [3, 0, 5, -1]
multiples = [scalars[i]*vectors[i] for i in range(4)]
multiples
```

```
[(6, 12, -9, 3, 6, 27), (0, 0, 0, 0, 0, 0),
 (-25, 10, 5, 5, -15, 0), (-3, -2, 5, -7, -1, -3)]
```

We have constructed two lists and used a list comprehension to just form the scalar multiple of each vector as part of the list `multiples`. Now we use the `sum()` function to add them all together.

```
sum(multiples)
```

```
(-22, 20, 1, 1, -10, 24)
```

We can improve on this in two ways. First, we can determine the number of elements in any list with the `len()` function. So we do not have to count up that we have 4 vectors (not that it is very hard to count!). Second, we can combine this all into one line, once we have defined the list of vectors and the list of scalars.

```
sum([scalars[i]*vectors[i] for i in range(len(vectors))])
```

```
(-22, 20, 1, 1, -10, 24)
```

The corresponding expression in mathematical notation, after a change of names and with counting starting from 1, would roughly be:

$$\sum_{i=1}^{4} a_i \mathbf{u}_i$$

Using `sum()` and a list comprehension might be overkill in this example, but we will find it very useful in just a minute.

SLC Solutions and Linear Combinations

We can easily illustrate Theorem SLSLC with Sage. We will use Archetype F as an example.

```
coeff = matrix(QQ, [[33, -16,  10,-2],
                    [99, -47,  27,-7],
                    [78, -36,  17,-6],
                    [-9,   2,   3, 4]])
const = vector(QQ, [-27, -77, -52, 5])
```

A solution to this system is $x_1 = 1$, $x_2 = 2$, $x_3 = -2$, $x_4 = 4$. So we will use these four values as scalars in a linear combination of the columns of the coefficient matrix. However, we do not have to type in the columns individually, we can have Sage extract them all for us into a list with the matrix method `.columns()`.

```
cols = coeff.columns()
cols
```

```
[(33, 99, 78, -9), (-16, -47, -36, 2),
 (10, 27, 17, 3), (-2, -7, -6, 4)]
```

With our scalars also in a list, we can compute the linear combination of the columns, like we did in Sage LC.

```
soln = [1, 2, -2, 4]
sum([soln[i]*cols[i] for i in range(len(cols))])
```

```
(-27, -77, -52, 5)
```

So we see that the solution gives us scalars that yield the vector of constants as a linear combination of the columns of the coefficient matrix. Exactly as predicted by Theorem SLSLC. We can duplicate this observation with just one line:

```
const == sum([soln[i]*cols[i] for i in range(len(cols))])
```

```
True
```

In a similar fashion we can test other potential solutions. With theory we will develop later, we will be able to determine that Archetype F has only one solution. Since Theorem SLSLC is an equivalence (Technique E), any other choice for the scalars should not create the vector of constants as a linear combination.

```
alt_soln = [-3, 2, 4, 1]
const == sum([alt_soln[i]*cols[i] for i in range(len(cols))])
```

```
False
```

Now would be a good time to find another system of equations, perhaps one with infinitely many solutions, and practice the techniques above.

SS2 Solving Systems, Part 2

We can now resolve a bit of the mystery around Sage's `.solve_right()` method. Recall from Sage SS1 that if a linear system has solutions, Sage only provides one solution, even in the case when there are infinitely many solutions. In our previous discussion, we used the system from Example ISSI.

```
coeff = matrix(QQ, [[ 1,  4,  0, -1,  0,   7, -9],
                    [ 2,  8, -1,  3,  9, -13,  7],
                    [ 0,  0,  2, -3, -4,  12, -8],
                    [-1, -4,  2,  4,  8, -31, 37]])
const = vector(QQ, [3, 9, 1, 4])
coeff.solve_right(const)
```

```
(4, 0, 2, 1, 0, 0, 0)
```

The vector **c** described in the statement of Theorem VFSLS is precisely the solution returned from Sage's `.solve_right()` method. This is the solution where we choose the α_i, $1 \leq i \leq n-r$ to all be zero, in other words, each free variable is set to zero (how convenient!). Free variables correspond to columns of the row-reduced augmented matrix that are *not* pivot columns. So we can profitably employ the `.nonpivots()` matrix method. Let us put this all together.

```
aug = coeff.augment(const)
reduced = aug.rref()
reduced
```

```
[ 1  4  0  0  2  1 -3  4]
[ 0  0  1  0  1 -3  5  2]
[ 0  0  0  1  2 -6  6  1]
[ 0  0  0  0  0  0  0  0]
```

```
aug.nonpivots()
```

```
(1, 4, 5, 6, 7)
```

Since the eighth column (numbered 7) of the reduced row-echelon form is not a pivot column, we know by Theorem RCLS that the system is consistent. We can use the indices of the remaining non-pivot columns to place zeros into the vector $\mathbf{c}$ in those locations. The remaining entries of $\mathbf{c}$ are the entries of the reduced row-echelon form in the last column, inserted in that order. Boom!

So we have three ways to get to the same solution: (a) row-reduce the augmented matrix and set the free variables all to zero, (b) row-reduce the augmented matrix and use the formula from Theorem VFSLS to construct $\mathbf{c}$, and (c) use Sage's `.solve_right()` method.

One mystery left to resolve. How can we get Sage to give us infinitely many solutions in the case of systems with an infinite solution set? This is best handled in the next section, Section SS, specifically in Sage SS3.

PSHS Particular Solutions, Homogeneous Solutions

Again, Sage is useful for illustrating a theorem, in this case Theorem PSPHS. We will illustrate both "directions" of this equivalence with the system from Example ISSI.

```
coeff = matrix(QQ,[[ 1,  4,  0, -1,  0,   7, -9],
                   [ 2,  8, -1,  3,  9, -13,  7],
                   [ 0,  0,  2, -3, -4,  12, -8],
                   [-1, -4,  2,  4,  8, -31, 37]])
n = coeff.ncols()
const = vector(QQ, [3, 9, 1, 4])
```

First we will build solutions to this system. Theorem PSPHS says we need a particular solution, i.e. one solution to the system, $\mathbf{w}$. We can get this from Sage's `.solve_right()` matrix method. Then for *any* vector $\mathbf{z}$ from the null space of the coefficient matrix, the new vector $\mathbf{y} = \mathbf{w} + \mathbf{z}$ should be a solution. We walk through this construction in the next few cells, where we have employed a specific element of the null space, $\mathbf{z}$, along with a check that it is really in the null space.

```
w = coeff.solve_right(const)
nsp = coeff.right_kernel(basis='pivot')
z = vector(QQ, [42, 0, 84, 28, -50, -47, -35])
z in nsp
```

```
True
```

```
y = w + z
y
```

```
(46, 0, 86, 29, -50, -47, -35)
```

```
const == sum([y[i]*coeff.column(i) for i in range(n)])
```

```
True
```

You can create solutions repeatedly via the creation of random elements of the null space. Be sure you have executed the cells above, so that `coeff`, `n`, `const`, `nsp` and `w` are all defined. Try executing the cell below repeatedly to test infinitely many solutions to the system. You can use the subsequent compute cell to peek at any of the solutions you create.

```
z = nsp.random_element()
y = w + z
const == sum([y[i]*coeff.column(i) for i in range(n)])
```

```
True
```

```
y     # random
```

```
(-11/2, 0, 45/2, 34, 0, 7/2, -2)
```

For the other direction, we present (and verify) two solutions to the linear system of equations. The condition that $\mathbf{y} = \mathbf{w} + \mathbf{z}$ can be rewritten as $\mathbf{y} - \mathbf{w} = \mathbf{z}$, where $\mathbf{z}$ is in the null space of the coefficient matrix. which of our two solutions is the "particular" solution and which is "some other" solution? It does not matter, it is all sematics at this point. What is important is that their *difference* is an element of the null space (in either order). So we define the solutions, along with checks that they are really solutions, then examine their difference.

```
soln1 = vector(QQ,[4, 0, -96, 29, 46, 76, 56])
const == sum([soln1[i]*coeff.column(i) for i in range(n)])
```

```
True
```

```
soln2 = vector(QQ,[-108, -84, 86, 589, 240, 283, 105])
const == sum([soln2[i]*coeff.column(i) for i in range(n)])
```

```
True
```

```
(soln1 - soln2) in nsp
```

```
True
```

Section SS: Spanning Sets

SS Spanning Sets

A strength of Sage is the ability to create infinite sets, such as the span of a set of vectors, from finite descriptions. In other words, we can take a finite set with just a handful of vectors and Sage will create the set that is the span of these vectors, which is an infinite set. Here we will show you how to do this, and show how you can use the results. The key command is the vector space method `.span()`.

```
V = QQ^4
v1 = vector(QQ, [1,1,2,-1])
v2 = vector(QQ, [2,3,5,-4])
W = V.span([v1, v2])
W
```

```
Vector space of degree 4 and dimension 2 over Rational Field
Basis matrix:
[ 1  0  1  1]
[ 0  1  1 -2]
```

```
x = 2*v1 + (-3)*v2
x
```

```
(-4, -7, -11, 10)
```

```
x in W
```

```
True
```

```
y = vector(QQ, [3, -1, 2, 2])
y in W
```

```
False
```

```
u = vector(QQ, [3, -1, 2, 5])
u in W
```

```
True
```

```
W <= V
```

```
True
```

Most of the above should be fairly self-explanatory, but a few comments are in order. The span, `W`, is created in the first compute cell with the `.span()` method, which accepts a list of vectors and needs to be employed as a method of a vector space. The information about `W` printed when we just input the span itself may be somewhat confusing, and as before, we need to learn some more theory to totally understand it all. For now you can see the number system (`Rational Field`) and

the number of entries in each vector (`degree 4`). The `dimension` may be more complicated than you first suspect.

Sets are all about membership, and we see that we can easily check membership in a span. We know the vector `x` will be in the span `W` since we built it as a linear combination of `v1` and `v2`. The vectors `y` and `u` are a bit more mysterious, but Sage can answer the membership question easily for both.

The last compute cell is something new. The symbol `<=` is meant here to be the "subset of" relation, i.e. a slight abuse of the mathematical symbol $\subseteq$, and then we are not surprised to learn that `W` really is a subset of `V`.

It is important to realize that the span is a *construction* that begins with a finite set, yet creates an infinite set. With a loop (the `for` command) and the `.random_element()` vector space method we can create *many*, but not all, of the elements of a span. In the examples below, you can edit the total number of random vectors produced, and you may need to click through to another file of output if you ask for more than about 100 vectors.

Each example is designed to illustrate some aspect of the behavior of the span command and to provoke some questions. So put on your mathematician's hat, evaluate the compute cells to create some sample elements, and then *study* the output carefully looking for patterns and maybe even conjecture some explanations for the behavior. The puzzle gets just a bit harder for each new example. (We use the `Sequence()` command to get nicely-formatted line-by-line output of the list, and notice that we are only providing a portion of the output here. You will want to evalaute the computation of `vecs` and then evaluate the next cell in the Sage notebook for maximum effect.)

```
V = QQ^4
W = V.span([ vector(QQ, [0, 1, 0, 1]),
             vector(QQ, [1, 0, 1, 0]) ])
vecs = [(i, W.random_element()) for i in range(100)]
```

```
Sequence(vecs, cr=True)          # random
```

```
[
(0, (1/5, 16, 1/5, 16)),
(1, (-3, 0, -3, 0)),
(2, (1/11, 0, 1/11, 0)),
...
(97, (-2, -1/2, -2, -1/2)),
(98, (1/13, -3, 1/13, -3)),
(99, (0, 1, 0, 1))
]
```

```
V = QQ^4
W = V.span([ vector(QQ, [0, 1, 1, 0]),
             vector(QQ, [0, 0, 1, 1]) ])
vecs = [(i, W.random_element()) for i in range(100)]
```

```
Sequence(vecs, cr=True)          # random
```

```
[
(0, (0, 1/9, 2, 17/9)),
(1, (0, -1/8, 3/2, 13/8)),
(2, (0, 1/2, -1, -3/2)),
...
(97, (0, 4/7, 24, 164/7)),
(98, (0, -5/2, 0, 5/2)),
(99, (0, 13/2, 1, -11/2))
]
```

```
V = QQ^4
W = V.span([ vector(QQ, [2, 1, 2, 1]),
             vector(QQ, [4, 2, 4, 2]) ])
vecs = [(i, W.random_element()) for i in range(100)]
```

```
Sequence(vecs, cr=True)          # random
```

```
[
(0, (1, 1/2, 1, 1/2)),
(1, (-1, -1/2, -1, -1/2)),
(2, (-1/7, -1/14, -1/7, -1/14)),
...
(97, (1/3, 1/6, 1/3, 1/6)),
(98, (0, 0, 0, 0)),
(99, (-11, -11/2, -11, -11/2))
]
```

```
V = QQ^4
W = V.span([ vector(QQ, [1, 0, 0, 0]),
             vector(QQ, [0, 1 ,0, 0]),
             vector(QQ, [0, 0, 1, 0]),
             vector(QQ, [0, 0, 0, 1]) ])
vecs = [(i, W.random_element()) for i in range(100)]
```

```
Sequence(vecs, cr=True)          # random
```

```
[
(0, (-7/4, -2, -1, 63)),
(1, (6, -2, -1/2, -28)),
(2, (5, -2, -2, -1)),
...
(97, (1, -1/2, -2, -1)),
(98, (-1/12, -4, 2, 1)),
(99, (2/3, 0, -4, -1))
]
```

```
V = QQ^4
W = V.span([ vector(QQ, [1, 2,  3, 4]),
             vector(QQ, [0,-1, -1, 0]),
             vector(QQ, [1, 1,  2, 4]) ])
vecs = [(i, W.random_element()) for i in range(100)]
```

```
Sequence(vecs, cr=True)          # random
```

```
[
(0, (-1, 3, 2, -4)),
(1, (-1/27, -1, -28/27, -4/27)),
(2, (-7/11, -1, -18/11, -28/11)),
...
(97, (1/3, -1/7, 4/21, 4/3)),
(98, (-1, -14, -15, -4)),
(99, (0, -2/7, -2/7, 0))
]
```

CSS Consistent Systems and Spans

With the notion of a span, we can expand our techniques for checking the consistency of a linear system. Theorem SLSLC tells us a system is consistent if and only if the vector of constants is a linear combination of the columns of the coefficient matrix. This is because Theorem SLSLC says that any solution to the system will provide a linear combination of the columns of the coefficient that equals the vector of constants. So consistency of a system is equivalent to the membership of the vector of constants in the span of the columns of the coefficient matrix. Read that last sentence again carefully. We will see this idea again, but more formally, in Theorem CSCS.

We will reprise Sage SLC, which is based on Archetype F. We again make use of the matrix method `.columns()` to get all of the columns into a list at once.

```
coeff = matrix(QQ, [[33, -16, 10, -2],
                    [99, -47, 27, -7],
                    [78, -36, 17, -6],
                    [-9,   2,  3,  4]])
column_list = coeff.columns()
column_list
```

```
[(33, 99, 78, -9), (-16, -47, -36, 2),
(10, 27, 17, 3), (-2, -7, -6, 4)]
```

```
span = (QQ^4).span(column_list)
const = vector(QQ, [-27, -77, -52, 5])
const in span
```

```
True
```

You could try to find an example of a vector of constants which would create an inconsistent system with this coefficient matrix. But there is no such thing. Here is why — the null space of `coeff` is trivial, just the zero vector.

```
nsp = coeff.right_kernel(basis='pivot')
nsp.list()
```

```
[(0, 0, 0, 0)]
```

The system is consistent, as we have shown, so we can apply Theorem PSPHS. We can read Theorem PSPHS as saying *any* two different solutions of the system will differ by an element of the null space, and the only possibility for this null space vector is just the zero vector. In other words, any two solutions *cannot* be different.

SSNS Spanning Sets for Null Spaces

We have seen that we can create a null space in Sage with the `.right_kernel()` method for matrices. We use the optional argument `basis='pivot'`, so we get exactly the spanning set $\{\mathbf{z}_1, \mathbf{z}_2, \mathbf{z}_3, \ldots, \mathbf{z}_{n-r}\}$ described in Theorem SSNS. This optional argument will overide Sage's default spanning set, whose purpose we will explain fully in Sage SUTH0. Here is Example SSNS again, along with a few extras we will comment on afterwards.

```
A = matrix(QQ, [[ 1,  3,  3, -1, -5],
                [ 2,  5,  7,  1,  1],
                [ 1,  1,  5,  1,  5],
                [-1, -4, -2,  0,  4]])
nsp = A.right_kernel(basis='pivot')
N = nsp.basis()
N
```

```
[
(-6, 1, 1, 0, 0),
(-4, 2, 0, -3, 1)
]
```

```
z1 = N[0]
z2 = N[1]
z = 4*z1 +(-3)*z2
z
```

```
(-12, -2, 4, 9, -3)
```

```
z in nsp
```

```
True
```

```
sum([z[i]*A.column(i) for i in range(A.ncols())])
```

```
(0, 0, 0, 0)
```

We built the null space as `nsp`, and then asked for its `.basis()`. For now, a "basis" will give us a spanning set, and with more theory we will understand it better. This is a set of vectors that form a spanning set for the null space, and with the `basis='pivot'` argument we have asked for the spanning set in the format described in Theorem SSNS. The spanning set `N` is a list of vectors, which we have extracted and named as `z1` and `z2`. The linear combination of these two vectors, named `z`, will also be in the null space since `N` is a spanning set for `nsp`. As verification, we have have used the five entries of `z` in a linear combination of the columns of `A`, yielding the zero vector (with four entries) as we expect.

We can also just ask Sage if `z` is in `nsp`:

```
z in nsp
```

```
True
```

Now is an appropriate time to comment on how Sage displays a null space when we just ask about it alone. Just as with a span, the number system and the number of entries are easy to see. Again, `dimension` should wait for a bit. But you will notice now that the `Basis matrix` has been replaced by `User basis matrix`. This is a consequence of our request for something other than the default basis (the `'pivot'` basis). As part of its standard information about a null space, or a span, Sage spits out the basis matrix. This is a matrix, whose *rows* are vectors in a spanning set. This matrix can be requested by itself, using the method `.basis_matrix()`. It is important to notice that this is very different than the output of `.basis()` which is a list of vectors. The two objects print very similarly, but even this is different — compare the organization of the brackets and parentheses. Finally the last command should print true for *any* span or null space Sage creates. If you rerun the commands below, be sure the null space `nsp` is defined from the code just above.

```
nsp
```

```
Vector space of degree 5 and dimension 2 over Rational Field
User basis matrix:
[-6  1  1  0  0]
[-4  2  0 -3  1]
```

```
nsp.basis_matrix()
```

```
[-6  1  1  0  0]
[-4  2  0 -3  1]
```

```
nsp.basis()
```

```
[
(-6, 1, 1, 0, 0),
(-4, 2, 0, -3, 1)
]
```

```
nsp.basis() == nsp.basis_matrix().rows()
```

```
True
```

SS3 Solving Systems, Part 3

We have deferred several mysteries and postponed some explanations of the terms Sage uses to describe certain objects. This is because Sage is a comprehensive tool that you can use throughout your mathematical career, and Sage assumes you already know a lot of linear algebra. Do not worry, you already know *some* linear algebra and will very soon will know a *whole lot more*. And we know enough now that we can now solve one mystery. How can we create *all* of the solutions to a linear system of equations when the solution set is infinite?

Theorem VFSLS described elements of a solution set as a single vector $\mathbf{c}$, plus linear combinations of vectors $\mathbf{u}_1$, $\mathbf{u}_2$, $\mathbf{u}_3$, ..., $\mathbf{u}_{n-r}$. The vectors $\mathbf{u}_j$ are identical to the vectors $\mathbf{z}_j$ in the description of the spanning set of a null space in Theorem SSNS, suitably adjusted from the setting of a general system and the relevant

homogeneous system for a null space. We can get the single vector $\mathbf{c}$ from the `.solve_right()` method of a coefficient matrix, once supplied with the vector of constants. The vectors $\{\mathbf{z}_1, \mathbf{z}_2, \mathbf{z}_3, \ldots, \mathbf{z}_{n-r}\}$ come from Sage in the basis returned by the `.right_kernel(basis='pivot')` method applied to the coefficient matrix. Theorem PSPHS amplifies and generalizes this discussion, making it clear that the choice of the particular particular solution $\mathbf{c}$ (Sage's choice, and your author's choice) is just a matter of convenience.

In our previous discussion, we used the system from Example ISSI. We will use it one more time.

```
coeff = matrix(QQ, [[ 1,  4,  0, -1,  0,   7, -9],
                    [ 2,  8, -1,  3,  9, -13,  7],
                    [ 0,  0,  2, -3, -4,  12, -8],
                    [-1, -4,  2,  4,  8, -31, 37]])
n = coeff.ncols()
const = vector(QQ, [3, 9, 1, 4])
c = coeff.solve_right(const)
c
```

```
(4, 0, 2, 1, 0, 0, 0)
```

```
N = coeff.right_kernel(basis='pivot').basis()
z1 = N[0]
z2 = N[1]
z3 = N[2]
z4 = N[3]
soln = c + 2*z1 - 3*z2 - 5*z3 + 8*z4
soln
```

```
(31, 2, -50, -71, -3, -5, 8)
```

```
check = sum([soln[i]*coeff.column(i) for i in range(n)])
check
```

```
(3, 9, 1, 4)
```

```
check == const
```

```
True
```

So we can describe the infinitely many solutions with just five items: the specific solution, $\mathbf{c}$, and the four vectors of the spanning set. Boom!

As an exercise in understanding Theorem PSPHS, you might begin with `soln` and add a linear combination of the spanning set for the null space to create another solution (which you can check).

Section LI: Linear Independence

LI Linear Independence

We can use the Sage tools we already have, along with Theorem LIVRN, to determine if sets are linearly independent. There is just one hitch — Sage has a

preference for placing vectors into matrices as *rows*, rather than as columns. When printing vectors on the screen, writing them *across* the screen makes good sense, and there are more mathematical reasons for this choice. But we have chosen to present introductory linear algebra with an emphasis on the *columns* of matrices — again, for good mathematical reasons. Fortunately, Sage allows us to build matrices from columns as well as rows.

Let us redo Example LDHS, the determination that a set of three vectors is linearly dependent. We will enter the vectors, construct a matrix with the vectors as *columns* via the `column_matrix()` constructor, and analyze. Here we go.

```
v1 = vector(QQ, [2, -1,  3,  4, 2])
v2 = vector(QQ, [6,  2, -1,  3, 4])
v3 = vector(QQ, [4,  3, -4, -1, 2])
A = column_matrix([v1, v2, v3])
A
```

```
[ 2  6  4]
[-1  2  3]
[ 3 -1 -4]
[ 4  3 -1]
[ 2  4  2]
```

```
A.ncols() == len(A.pivots())
```

```
False
```

Notice that we never explicitly row-reduce `A`, though this computation must happen behind the scenes when we compute the list of pivot columns. We do not really care *where* the pivots are (the actual list), but rather we want to know *how many* there are, thus we ask about the *length* of the list with the function `len()`. Once we construct the matrix, the analysis is quick. With $n \neq r$, Theorem LIVRN tells us the set is linearly dependent.

Reprising Example LIS with Sage would be good practice at this point. Here is an empty compute cell to use.

While it is extremely important to understand the approach outlined above, Sage has a convenient tool for working with linear independence. `.linear_dependence()` is a method for vector spaces, which we feed in a list of vectors. The output is again a list of vectors, each one containing the scalars that yield a nontrivial relation of linear dependence on the input vectors. We will give this method a workout in the next section, but for now we are interested in the case of a linearly independent set. In this instance, the method will return nothing (an empty list, really). Not even the all-zero vector is produced, since it is not interesting and definitely is not surprising.

Again, we will not say anymore about the output of this method until the next section, and do not let its use replace a good conceptual understanding of this section. We will redo Example LDHS again, you try Example LIS again. If you are playing along, be sure `v1, v2, v3` are defined from the code above.

```
L = [v1, v2, v3]
V = QQ^5
V.linear_dependence(L) == []
```

```
False
```

The only comment to make here is that we need to create the vector space `QQ^5` since `.linear_dependence()` is a method of vector spaces. Example LIS should proceed similarly, though being a linearly independent set, the comparison with the empty list should yield `True`.

NME2 Nonsingular Matrices, Round 2

As will be our habit, we can illustrate properties of nonsingular matrices with random square matrices. Review Sage NME1 for a refresher on generating these matrices. Here we will illustrate the fifth condition, half of Theorem NMLIC.

```
n = 6
A = random_matrix(QQ, n, algorithm='unimodular')
A                                   # random
```

```
[   2    7   37  -79  268   44]
[  -1   -3  -16   33 -110  -16]
[   1    1    7  -14   44    5]
[   0   -3  -15   34 -118  -23]
[   2    6   33  -68  227   34]
[  -1   -3  -16   35 -120  -21]
```

```
V = QQ^n
V.linear_dependence(A.columns()) == []
```

```
True
```

You should always get an empty list (`[]`) as the result of the second compute cell, no matter which random (unimodular) matrix you produce prior. Note that the list of vectors created using `.columns()` is exactly the list we want to feed to `.linear_dependence()`.

LISS Linearly Independent Spanning Sets

No new commands here, but instead we can now reveal and explore one of our Sage mysteries. When we create a null space with `.right_kernel(basis='pivot')`, or a span with `.span()`, Sage internally creates a new spanning set for the null space or span. We see the vectors of these spanning sets as the rows of the `Basis matrix` or `User basis matrix` when we print one of these sets.

But more than a spanning set, these vectors are always a linearly independent set. The advantages of this will have to wait for more theory, but for now, this may explain why the spanning set changes (and sometimes shrinks). We can also demonstrate this behavior, by creating the span of a large set of (linearly dependent) vectors.

```
V = QQ^5
v1 = vector(QQ, [ 5,   2, -11, -24, -17])
v2 = vector(QQ, [ 4,  13,  -5,  -4,  13])
v3 = vector(QQ, [ 8,  29,  -9,  -4,  33])
v4 = vector(QQ, [ 1,   1,  -2,  -4,  -2])
v5 = vector(QQ, [-7, -25,   8,   4, -28])
v6 = vector(QQ, [ 0,  -3,  -1,  -4,  -7])
v7 = vector(QQ, [-1,   2,   3,   8,   9])
```

```
L = [v1, v2, v3, v4, v5, v6, v7]
V.linear_dependence(L) == []
```

```
False
```

```
W = V.span(L)
W
```

```
Vector space of degree 5 and dimension 2 over Rational Field
Basis matrix:
[    1     0  -7/3 -16/3 -13/3]
[    0     1   1/3   4/3   7/3]
```

```
S = W.basis()
S
```

```
[
(1, 0, -7/3, -16/3, -13/3),
(0, 1, 1/3, 4/3, 7/3)
]
```

```
X = V.span(S)
X == W
```

```
True
```

```
V.linear_dependence(S) == []
```

```
True
```

So the above examines properties of the two vectors, `S`, that Sage computes for the span `W` built from the seven vectors in the set `L`. We see that the span of `S` is equal to the span of `L` via the comparison `X == W`. And the smaller set, `S`, is linearly independent, as promised. Notice that we do not need to use Sage to know that `L` is linearly dependent, this is guaranteed by Theorem MVSLD.

Section LDS: Linear Dependence and Spans

RLD Relations of Linear Dependence

Example RSC5 turned on a nontrivial relation of linear dependence (Definition RLDCV) on the set $\{\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3, \mathbf{v}_4\}$. Besides indicating linear independence, the Sage vector space method `.linear_dependence()` produces relations of linear dependence for linearly dependent sets. Here is how we would employ this method in Example RSC5. The optional argument `zeros='right'` will produce results consistent with our work here, you can also experiment with `zeros='left'` (which is the default).

```
V = QQ^5
v1 = vector(QQ, [1,  2, -1,   3,  2])
v2 = vector(QQ, [2,  1,  3,   1,  2])
v3 = vector(QQ, [0, -7,  6, -11, -2])
v4 = vector(QQ, [4,  1,  2,   1,  6])
R = [v1, v2, v3, v4]
```

```
L = V.linear_dependence(R, zeros='right')
L[0]
```

```
(-4, 0, -1, 1)
```

```
-4*v1 + 0*v2 +(-1)*v3 +1*v4
```

```
(0, 0, 0, 0, 0)
```

```
V.span(R) == V.span([v1, v2, v4])
```

```
True
```

You can check that the list `L` has just one element (maybe with `len(L)`), but realize that any multiple of the vector `L[0]` is also a relation of linear dependence on `R`, most of which are nontrivial. Notice that we have verified the final conclusion of Example RSC5 with a comparison of two spans.

We will give the `.linear_dependence()` method a real workout in the next Sage subsection (Sage COV) — this is just a quick introduction.

COV Casting Out Vectors

We will redo Example COV, though somewhat tersely, just producing the justification for each time we toss a vector (a specific relation of linear dependence), and then verifying that the resulting spans, each with one fewer vector, still produce the original span. We also introduce the `.remove()` method for lists. Ready? Here we go.

```
V = QQ^4
v1 = vector(QQ,  [ 1,   2,  0,  -1])
v2 = vector(QQ,  [ 4,   8,  0,  -4])
v3 = vector(QQ,  [ 0,  -1,  2,   2])
v4 = vector(QQ,  [-1,   3, -3,   4])
v5 = vector(QQ,  [ 0,   9, -4,   8])
v6 = vector(QQ,  [ 7, -13, 12, -31])
v7 = vector(QQ,  [-9,   7, -8,  37])
S = [v1, v2, v3, v4, v5, v6, v7]
W = V.span(S)
D = V.linear_dependence(S, zeros='right')
D
```

```
[
(-4, 1, 0, 0, 0, 0, 0),
(-2, 0, -1, -2, 1, 0, 0),
(-1, 0, 3, 6, 0, 1, 0),
(3, 0, -5, -6, 0, 0, 1)
]
```

```
D[0]
```

```
(-4, 1, 0, 0, 0, 0, 0)
```

```
S.remove(v2)
W == V.span(S)
```

```
True
```

```
D[1]
```

```
(-2, 0, -1, -2, 1, 0, 0)
```

```
S.remove(v5)
W == V.span(S)
```

```
True
```

```
D[2]
```

```
(-1, 0, 3, 6, 0, 1, 0)
```

```
S.remove(v6)
W == V.span(S)
```

```
True
```

```
D[3]
```

```
(3, 0, -5, -6, 0, 0, 1)
```

```
S.remove(v7)
W == V.span(S)
```

```
True
```

```
S
```

```
[(1, 2, 0, -1), (0, -1, 2, 2), (-1, 3, -3, 4)]
```

```
S == [v1, v3, v4]
```

```
True
```

Notice that `S` begins with all seven original vectors, and slowly gets whittled down to just the list `[v1, v3, v4]`. If you experiment with the above commands, be sure to return to the start and work your way through in order, or the results will not be right.

As a bonus, notice that the set of relations of linear dependence provided by Sage, `D`, is itself a linearly independent set (but within a very different vector space). Is that too weird?

```
U = QQ^7
U.linear_dependence(D) == []
```

```
True
```

Now, can you answer the extra credit question from Example COV using Sage?

RS Reducing a Span

Theorem BS allows us to construct a reduced spanning set for a span. As with the theorem, employing Sage we begin by constructing a matrix with the vectors of the spanning set as columns. Here is a do-over of Example RSC4, illustrating the use of Theorem BS in Sage.

```
V = QQ^4
v1 = vector(QQ, [1,1,2,1])
v2 = vector(QQ, [2,2,4,2])
v3 = vector(QQ, [2,0,-1,1])
v4 = vector(QQ, [7,1,-1,4])
v5 = vector(QQ, [0,2,5,1])
S = [v1, v2, v3, v4, v5]
A = column_matrix(S)
T = [A.column(p) for p in A.pivots()]
T
```

```
[(1, 1, 2, 1), (2, 0, -1, 1)]
```

```
V.linear_dependence(T) == []
```

```
True
```

```
V.span(S) == V.span(T)
```

```
True
```

Notice how we compute `T` with the single line that mirrors the construction of the set $T = \{\mathbf{v}_{d_1}, \mathbf{v}_{d_2}, \mathbf{v}_{d_3}, \ldots \mathbf{v}_{d_r}\}$ in the statement of Theorem BS. Again, the row-reducing is hidden in the use of the `.pivot()` matrix method, which necessarily must compute the reduced row-echelon form. The final two compute cells verify both conclusions of the theorem.

RES Reworking a Span

As another demonstration of using Sage to help us understand spans, linear combinations, linear independence and reduced row-echelon form, we will recreate parts of Example RES. Most of this should be familiar, but see the commments following.

```
V = QQ^4
v1 = vector(QQ, [2,1,3,2])
v2 = vector(QQ, [-1,1,0,1])
v3 = vector(QQ, [-8,-1,-9,-4])
v4 = vector(QQ, [3,1,-1,-2])
v5 = vector(QQ, [-10,-1,-1,4])
y = 6*v1 - 7*v2 + v3 +6*v4 + 2*v5
```

```
y
```

```
(9, 2, 1, -3)
```

```
R = [v1, v2, v3, v4, v5]
X = V.span(R)
y in X
```

```
True
```

```
A = column_matrix(R)
P = [A.column(p) for p in A.pivots()]
W = V.span(P)
W == X
```

```
True
```

```
y in W
```

```
True
```

```
coeff = column_matrix(P)
coeff.solve_right(y)
```

```
(1, -1, 2)
```

```
coeff.right_kernel()
```

```
Vector space of degree 3 and dimension 0 over Rational Field
Basis matrix:
[]
```

```
V.linear_dependence(P) == []
```

```
True
```

The final two results — a trivial null space for `coeff` and the linear independence of `P` — both individually imply that the solution to the system of equations (just prior) is unique. Sage produces its own linearly independent spanning set for each span, as we see whenever we inquire about a span.

```
X
```

```
Vector space of degree 4 and dimension 3 over Rational Field
Basis matrix:
[    1     0     0 -8/15]
[    0     1     0  7/15]
[    0     0     1 13/15]
```

Can you extract the three vectors that Sage uses to span `X` and solve the appropriate system of equations to see how to write `y` as a linear combination of these three vectors? Once you have done that, check your answer *by hand* and think about how using Sage could have been overkill for this question.

Section O: Orthogonality

EVIC Exact Versus Inexact Computations

We are now at a crossroads in our use of Sage. So far our computations have involved rational numbers: fractions of two integers. Sage is able to work with integers of seemingly unlimited size, and then can work with rational numbers exactly. So all of our computations have been exactly correct so far. In practice, many computations, especially those that originate with data, are not so precise. Then we *represent* real numbers by "floating point numbers." Since the real numbers are infinite, finite computers must fake it with an extremely large, but still finite, collection of numbers. The price we pay is that some computations will be just slightly imprecise when there is no number available that represents the exact answer.

You should now appreciate two problems that occur. If we were to row-reduce a matrix with floating point numbers, there are potentially many computations and if a small amount of imprecision arises in each one, these errors can accumulate and lead to wildly incorrect answers. When we row-reduce a matrix, whether or not an entry is zero or not is critically important in the decisions we make about which row operation to perform. If we have an extremely small number (like 10^{-16}) how can we be sure if it is zero or not?

Why discuss this now? What is $\alpha = \sqrt{\frac{7}{3}}$? Hard to say exactly, but it is definitely not a rational number. Norms of vectors will feature prominently in all our discussions about orthogonal vectors, so we now have to recognize the need to work with square roots properly. We have two strategies in Sage.

The number system `QQbar`, also known as the "field of algebraic numbers," is a truly amazing feature of Sage. It contains the rational numbers, plus *every* root of *every* polynomial with coefficients that are rational numbers. For example, notice that α above is one solution to the polynomial equation $3x^2 - 7 = 0$ and thus is a number in `QQbar`, so Sage can work with it *exactly*. These numbers are called "algebraic numbers" and you can recognize them since they print with a question mark near the end to remind you that when printed as a decimal they are approximations of numbers that Sage carries internally as exact quantities. For example α can be created with `QQbar(sqrt(7/3))` and will print as `1.527525231651947?`. Notice that complex numbers begin with the introduction of the imaginary number i, which is a root of the polynomial equation $x^2 + 1 = 0$, so the field of algebraic numbers contains many complex numbers. The downside of `QQbar` is that computations are slow (relatively speaking), so this number system is most useful for examples and demonstrations.

The other strategy is to work strictly with approximate numbers, cognizant of the potential for inaccuracies. Sage has two such number systems: `RDF` and `CDF`, which are comprised of "double precision" floating point numbers, first limited to just the reals, then expanded to the complexes. Double-precision refers to the use of 64 bits to store the sign, mantissa and exponent in the representation of a real number. This gives 53 bits of precision. Do not confuse these fields with `RR` and `CC`, which are similar in appearance but very different in implementation. Sage has implementations of several computations designed exclusively for `RDF` and `CDF`, such as the norm. And they are very, very fast. But some computations, like echelon form, can be wildly unreliable with these approximate numbers. We will have more

to say about this as we go. In practice, you can use `CDF`, since `RDF` is a subset and only different in very limited cases.

In summary, `QQbar` is an extension of `QQ` which allows exact computations, but can be slow for large examples. `RDF` and `CDF` are fast, with special algorithms to control much of the imprecision in some, but not all, computations. So we need to be vigilant and skeptical when we work with these approximate numbers. We will use both strategies, as appropriate.

CNIP Conjugates, Norms and Inner Products

Conjugates, of complex numbers and of vectors, are straightforward, in `QQbar` or in `CDF`.

```
alpha = QQbar(2 + 3*I)
alpha.conjugate()
```

```
2 - 3*I
```

```
beta = CDF(2+3*I)
beta.conjugate()
```

```
2.0 - 3.0*I
```

```
v = vector(QQbar, [5-3*I, 2+6*I])
v.conjugate()
```

```
(5 + 3*I, 2 - 6*I)
```

```
w = vector(CDF, [5-3*I, 2+6*I])
w.conjugate()
```

```
(5.0 + 3.0*I, 2.0 - 6.0*I)
```

The term "inner product" means slightly different things to different people. For some, it is the "dot product" that you may have seen in a calculus or physics course. Our inner product could be called the "Hermitian inner product" to emphasize the use of vectors over the complex numbers and conjugating some of the entries. So Sage has a `.dot_product()`, `.inner_product()`, and `.hermitian_inner_product()` — we want to use the last one.

From now on, when we mention an inner product in the context of using Sage, we will mean `.hermitian_inner_product()`. We will redo the first part of Example CSIP. Notice that the syntax is a bit asymmetric.

```
u = vector(QQbar, [2+3*I,  5+2*I, -3+I])
v = vector(QQbar, [1+2*I, -4+5*I,  5*I])
u.hermitian_inner_product(v)
```

```
3 + 19*I
```

Norms are as easy as conjugates. Easier maybe. It might be useful to realize that Sage uses entirely distinct code to compute an exact norm over `QQbar` versus an approximate norm over `CDF`, though that is totally transparent as you issue commands. Here is Example CNSV reprised.

```
entries = [3+2*I, 1-6*I, 2+4*I, 2+I]
u = vector(QQbar, entries)
u.norm()
```

```
8.660254037844397
```

```
u = vector(CDF, entries)
u.norm()
```

```
8.660254037844
```

```
numerical_approx(5*sqrt(3), digits = 30)
```

```
8.66025403784438646763723170753
```

We have three different numerical approximations, the latter 30-digit number being an approximation to the answer in the text. But there is no inconsistency between them. The first, an algebraic number, is represented internally as $5 * a$ where a is a root of the polynomial equation $x^2 - 3 = 0$, in other words it is $5\sqrt{3}$. The `CDF` value prints with a few digits less than what is carried internally. Notice that our different definitions of the inner product make no difference in the computation of a norm.

One warning now that we are working with complex numbers. It is easy to "clobber" the symbol `I` used for the imaginary number i. In other words, Sage will allow you to assign it to something else, rendering it useless. An identity matrix is a likely reassignment. If you run the next compute cell, be sure to evaluate the compute cell afterward to restore `I` to its usual role.

```
alpha = QQbar(5 - 6*I)
I = identity_matrix(2)
beta = QQbar(2+5*I)
```

```
Traceback (most recent call last):
...
TypeError: Illegal initializer for algebraic number
```

```
restore()
I^2
```

```
-1
```

We will finish with a verification of Theorem IPN. To test equality it is best if we work with entries from `QQbar`.

```
v = vector(QQbar, [2-3*I, 9+5*I, 6+2*I, 4-7*I])
v.hermitian_inner_product(v) == v.norm()^2
```

```
True
```

OGS Orthogonality and Gram-Schmidt

It is easy enough to check a pair of vectors for orthogonality (is the inner product zero?). To check that a set is orthogonal, we just need to do this repeatedly. This is a redo of Example AOS.

```
x1 = vector(QQbar, [    1+I,       1,      1-I,       I])
x2 = vector(QQbar, [  1+5*I,   6+5*I,     -7-I,   1-6*I])
x3 = vector(QQbar, [-7+34*I, -8-23*I, -10+22*I, 30+13*I])
x4 = vector(QQbar, [ -2-4*I,     6+I,    4+3*I,     6-I])
S = [x1, x2, x3, x4]
ips = [S[i].hermitian_inner_product(S[j])
                  for i in range(3) for j in range(i+1,3)]
all([ip == 0 for ip in ips])
```

```
True
```

Notice how the list comprehension computes each pair just once, and never checks the inner product of a vector with itself. If we wanted to check that a set is orthonormal, the "normal" part is less involved. We will check the set above, even though we can clearly see that the four vectors are not even close to being unit vectors. Be sure to run the above definitions of `S` before running the next compute cell.

```
ips = [S[i].hermitian_inner_product(S[i]) for i in range(3)]
all([ip == 1 for ip in ips])
```

```
False
```

Applying the Gram-Schmidt procedure to a set of vectors is the type of computation that a program like Sage is perfect for. Gram-Schmidt is implemented as a method for matrices, where we interpret the rows of the matrix as the vectors in the original set. The result is two matrices, where the first has rows that are the orthogonal vectors. The second matrix has rows that provide linear combinations of the orthogonal vectors that equal the original vectors. The original vectors do not need to form a linearly independent set, and when the set is linearly dependent, then zero vectors produced are not part of the returned set.

Over `CDF` the set is automatically orthonormal, and since a different algorithm is used (to help control the imprecisions), the results will look different than what would result from Theorem GSP. We will illustrate with the vectors from Example GSTV.

```
v1 = vector(CDF, [ 1, 1+I,   1])
v2 = vector(CDF, [-I,   1, 1+I])
v3 = vector(CDF, [ 0,   I,   I])
A = matrix([v1,v2,v3])
G, M = A.gram_schmidt()
G.round(5)
```

```
[                -0.5          -0.5 - 0.5*I                  -0.5]
[ 0.30151 + 0.45227*I -0.15076 + 0.15076*I -0.30151 - 0.75378*I]
[   0.6396 + 0.2132*I   -0.2132 - 0.6396*I    0.2132 + 0.2132*I]
```

We formed the matrix A with the three vectors as rows, and of the two outputs we are interested in the first one, whose rows form the orthonormal set. We round the numbers to 5 digits, just to make the result fit nicely on your screen. Let us do it again, now exactly over `QQbar`. We will output the entries of the matrix as list, working across rows first, so it fits nicely.

```
v1 = vector(QQbar, [ 1, 1+I,   1])
v2 = vector(QQbar, [-I,   1, 1+I])
v3 = vector(QQbar, [ 0,   I,   I])
A = matrix([v1,v2,v3])
G, M = A.gram_schmidt(orthonormal=True)
Sequence(G.list(), cr=True)
```

```
[
0.50000000000000000?,
0.50000000000000000? + 0.50000000000000000?*I,
0.50000000000000000?,
-0.30151134457776360? - 0.45226701686664540?*I,
0.15075567228888180? - 0.15075567228888180?*I,
0.30151134457776360? + 0.75377836144440910?*I,
-0.63960214906683130? - 0.21320071635561050?*I,
0.21320071635561050? + 0.63960214906683130?*I,
-0.21320071635561050? - 0.21320071635561050?*I
]
```

Notice that we asked for orthonormal output, so the rows of `G` are the vectors $\{\mathbf{w}_1, \mathbf{w}_2, \mathbf{w}_3\}$ in Example ONTV. Exactly. We can restrict ourselves to `QQ` and forego the "normality" to obtain just the orthogonal set $\{\mathbf{u}_1, \mathbf{u}_2, \mathbf{u}_3\}$ of Example GSTV.

```
v1 = vector(QQbar, [ 1, 1+I,   1])
v2 = vector(QQbar, [-I,   1, 1+I])
v3 = vector(QQbar, [ 0,   I,   I])
A = matrix([v1, v2, v3])
G, M = A.gram_schmidt(orthonormal=False)
Sequence(G.list(), cr=True)
```

```
[
1,
I + 1,
1,
-0.50000000000000000? - 0.75000000000000000?*I,
0.25000000000000000? - 0.25000000000000000?*I,
0.50000000000000000? + 1.25000000000000000?*I,
-0.27272727272727280? - 0.09090909090909090?*I,
0.09090909090909090? + 0.27272727272727280?*I,
-0.09090909090909090? - 0.09090909090909090?*I
]
```

Notice that it is an error to ask for an orthonormal set over `QQ` since you cannot expect to take square roots of rationals and stick with rationals.

```
v1 = vector(QQ, [1, 1])
v2 = vector(QQ, [2, 3])
A = matrix([v1,v2])
G, M = A.gram_schmidt(orthonormal=True)
```

```
Traceback (most recent call last):
...
TypeError: QR decomposition unable to compute square roots in Rational Field
```

Chapter M
Matrices

Section MO: Matrix Operations

MS Matrix Spaces

Sage defines our set M_{mn} as a "matrix space" with the command `MatrixSpace(R, m, n)` where `R` is a number system and `m` and `n` are the number of rows and number of columns, respectively. This object does not have much functionality defined in Sage. Its main purposes are to provide a parent for matrices, and to provide another way to create matrices. The two matrix operations just defined (addition and scalar multiplication) are implemented as you would expect. In the example below, we create two matrices in $M_{2,3}$ from just a list of 6 entries, by coercing the list into a matrix by using the relevant matrix space as if it were a function. Then we can perform the basic operations of matrix addition (Definition MA) and matrix scalar multiplication (Definition MSM).

```
MS = MatrixSpace(QQ, 2, 3)
MS
```

```
Full MatrixSpace of 2 by 3 dense matrices over Rational Field
```

```
A = MS([1, 2, 1, 4, 5, 4])
B = MS([1/1, 1/2, 1/3, 1/4, 1/5, 1/6])
A + B
```

```
[   2  5/2  4/3]
[17/4 26/5 25/6]
```

```
60*B
```

```
[60 30 20]
[15 12 10]
```

```
5*A - 120*B
```

```
[-115  -50  -35]
[ -10    1    0]
```

Coercion can make some interesting conveniences possible. Notice how the scalar 37 in the following expression is coerced to 37 times an identity matrix of the proper size.

```
A = matrix(QQ, [[ 0,  2, 4],
                [ 6,  0, 8],
                [10, 12, 0]])
A + 37
```

```
[37  2  4]
[ 6 37  8]
[10 12 37]
```

This coercion only applies to sums with *square* matrices. You might try this again, but with a rectangular matrix, just to see what the error message says.

MO Matrix Operations

Every operation in this section is implemented in Sage. The only real subtlety is determining if certain matrices are symmetric, which we will discuss below. In linear algebra, the term "adjoint" has two unrelated meanings, so you need to be careful when you see this term. In particular, in Sage it is used to mean something different. So our version of the adjoint is implemented as the matrix method `.conjugate_transpose()`. Here are some straightforward examples.

```
A = matrix(QQ, [[-1, 2, 4],
                [ 0, 3, 1]])
A
```

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

```
A.transpose()
```

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

```
A.is_symmetric()
```

```
False
```

```
B = matrix(QQ, [[ 1, 2, -1],
                [ 2, 3,  4],
                [-1, 4, -6]])
B.is_symmetric()
```

```
True
```

```
C = matrix(QQbar, [[  2-I, 3+4*I],
                   [5+2*I,     6]])
C.conjugate()
```

```
[2 + 1*I 3 - 4*I]
[5 - 2*I       6]
```

```
C.conjugate_transpose()
```

```
[2 + 1*I 5 - 2*I]
[3 - 4*I       6]
```

With these constructions, we can test, or demonstrate, some of the theorems above. Of course, this does not make the theorems true, but is satisfying nonetheless. This can be an effective technique when you are learning new Sage commands or new linear algebra — if your computations are not consistent with theorems, then your understanding of the linear algebra may be flawed, or your understanding of Sage may be flawed, or Sage may have a bug! Note in the following how we use comparison (==) between matrices as an implementation of matrix equality (Definition ME).

```
A = matrix(QQ, [[ 1, -1, 3],
                [-3,  2, 0]])
B = matrix(QQ, [[5, -2,  7],
                [1,  3, -2]])
C = matrix(QQbar, [[2+3*I, 1 - 6*I], [3, 5+2*I]])
A.transpose().transpose() == A
```

```
True
```

```
(A+B).transpose() == A.transpose() + B.transpose()
```

```
True
```

```
(2*C).conjugate() == 2*C.conjugate()
```

```
True
```

```
a = QQbar(3 + 4*I)
acon = a.conjugate()
(a*C).conjugate_transpose() == acon*C.conjugate_transpose()
```

```
True
```

The opposite is true — you can use theorems to convert, or express, Sage code into alternative, but mathematically equivalent forms.

Here is the subtlety. With approximate numbers, such as in RDF and CDF, it can be tricky to decide if two numbers are equal, or if a very small number is zero or not. In these situations Sage allows us to specify a "tolerance" — the largest number that can be effectively considered zero. Consider the following:

```
A = matrix(CDF, [[1.0, 0.0], [0.0, 1.0]])
A
```

```
[1.0 0.0]
[0.0 1.0]
```

```
A.is_symmetric()
```

```
True
```

```
A[0,1] = 0.000000000002
A
```

```
[  1.0 2e-12]
[  0.0   1.0]
```

```
A.is_symmetric()
```

```
False
```

```
A[0,1] = 0.000000000001
A
```

```
[  1.0 1e-12]
[  0.0   1.0]
```

```
A.is_symmetric()
```

```
True
```

Clearly the last result is not correct. This is because $0.000000000001 = 1.0 \times 10^{-12}$ is "small enough" to be confused as equal to the zero in the other corner of the matrix. However, Sage will let us set our own idea of when two numbers are equal, by setting a tolerance on the difference between two numbers that will allow them to be considered equal. The default tolerance is set at 1.0×10^{-12}. Here we use Sage's syntax for scientific notation to specify the tolerance.

```
A = matrix(CDF, [[1.0, 0.0], [0.0, 1.0]])
A.is_symmetric()
```

```
True
```

```
A[0,1] = 0.000000000001
A.is_symmetric()
```

```
True
```

```
A.is_symmetric(tol=1.0e-13)
```

```
False
```

This is not a course in numerical linear algebra, even if that is a fascinating field of study. To concentrate on the main ideas of introductory linear algebra, whenever possible we will concentrate on number systems like the rational numbers or algebraic numbers where we can rely on exact results. If you are ever unsure if a number system is exact or not, just ask.

```
QQ.is_exact()
```

```
True
```

```
RDF.is_exact()
```

```
False
```

Section MM: Matrix Multiplication

MVP Matrix-Vector Product

A matrix-vector product is very natural in Sage, and we can check the result against a linear combination of the columns.

```
A = matrix(QQ, [[1, -3,  4,  5],
                [2,  3, -2,  0],
                [5,  6,  8, -2]])
v = vector(QQ, [2, -2, 1, 3])
A*v
```

```
(27, -4, 0)
```

```
sum([v[i]*A.column(i) for i in range(len(v))])
```

```
(27, -4, 0)
```

Notice that when a matrix is square, a vector of the correct size can be used in Sage in a product with a matrix by placing the vector on either side of the matrix. However, the two results are *not* the same, and we will not have ocassion to place the vector on the left any time soon. So, despite the possibility, be certain to keep your vectors on the right side of a matrix in a product.

```
B = matrix(QQ, [[ 1, -3,  4,  5],
                [ 2,  3, -2,  0],
                [ 5,  6,  8, -2],
                [-4,  1,  1,  2]])
w = vector(QQ, [1, 2, -3, 2])
B*w
```

```
(-7, 14, -11, -1)
```

```
w*B
```

```
(-18, -13, -22, 15)
```

```
B*w == w*B
```

```
False
```

Since a matrix-vector product forms a linear combination of columns of a matrix, it is now very easy to check if a vector is a solution to a system of equations. This is basically the substance of Theorem SLEMM. Here we construct a system of equations and construct two solutions and one non-solution by applying Theorem PSPHS. Then we use a matrix-vector product to verify that the vectors are, or are not, solutions.

```
coeff = matrix(QQ, [[-1,  3, -1, -1,  0,  2],
                    [ 2, -6,  1, -2, -5, -8],
                    [ 1, -3,  2,  5,  4,  1],
                    [ 2, -6,  2,  2,  1, -3]])
const = vector(QQ, [13, -25, -17, -23])
solution1 = coeff.solve_right(const)
coeff*solution1
```

```
(13, -25, -17, -23)
```

```
nsp = coeff.right_kernel(basis='pivot')
nsp
```

```
Vector space of degree 6 and dimension 3 over Rational Field
User basis matrix:
[ 3  1  0  0  0  0]
[ 3  0 -4  1  0  0]
[ 1  0  1  0 -1  1]
```

```
nspb = nsp.basis()
solution2 = solution1 + 5*nspb[0]+(-4)*nspb[1]+2*nspb[2]
coeff*solution2
```

```
(13, -25, -17, -23)
```

```
nonnullspace = vector(QQ, [5, 0, 0, 0, 0, 0])
nonnullspace in nsp
```

```
False
```

```
nonsolution = solution1 + nonnullspace
coeff*nonsolution
```

```
(8, -15, -12, -13)
```

We can now explain the difference between "left" and "right" variants of various Sage commands for linear algebra. Generally, the direction refers to *where the vector is placed* in a matrix-vector product. We place a vector on the right and understand

this to mean a linear combination of the columns of the matrix. Placing a vector to the left of a matrix can be understood, in a manner totally consistent with our upcoming definition of matrix multiplication, as a linear combination of the *rows* of the matrix.

So the difference between `A.solve_right(v)` and `A.solve_left(v)` is that the former asks for a vector `x` such that `A*x == v`, while the latter asks for a vector `x` such that `x*A == v`. Given Sage's preference for rows, a direction-neutral version of a command, if it exists, will be the "left" version. For example, there is a `.right_kernel()` matrix method, while the `.left_kernel()` and `.kernel()` methods are identical — the names are synonyms for the exact same routine.

So when you see a Sage command that comes in "left" and "right" variants figure out just what part of the defined object involves a matrix-vector product and form an interpretation from that.

MM Matrix Multiplication

Matrix multiplication is very natural in Sage, and is just as easy as multiplying two numbers. We illustrate Theorem EMP by using it to compute the entry in the first row and third column.

```
A = matrix(QQ, [[3, -1, 2,  5],
                [9,  1, 2, -4]])
B = matrix(QQ, [[1,  6, 1],
                [0, -1, 2],
                [5,  2, 3],
                [1,  1, 1]])
A*B
```

```
[18 28 12]
[15 53 13]
```

```
sum([A[0,k]*B[k,2] for k in range(A.ncols())])
```

```
12
```

Note in the final statement, we could replace `A.ncols()` by `B.nrows()` since these two quantities must be identical. You can experiment with the last statement by editing it to compute any of the five other entries of the matrix product.

Square matrices can be multiplied in either order, but the result will almost always be different. Execute repeatedly the following products of two random 4×4 matrices, with a check on the equality of the two products in either order. It is possible, but highly unlikely, that the two products will be equal. So if this compute cell ever produces `True` it will be a minor miracle.

```
A = random_matrix(QQ,4,4)
B = random_matrix(QQ,4,4)
A*B == B*A       # random, sort of
```

```
False
```

PMM Properties of Matrix Multiplication

As before, we can use Sage to demonstrate theorems. With randomly-generated matrices, these verifications might be even more believable. Some of the above results

should feel fairly routine, but some are perhaps contrary to intuition. For example, Theorem MMT might at first glance seem surprising due to the requirement that the order of the product is reversed. Here is how we would investigate this theorem in Sage. The following compute cell should *always* return `True`. Repeated experimental evidence does not make a proof, but certainly gives us confidence.

```
A = random_matrix(QQ, 3, 7)
B = random_matrix(QQ, 7, 5)
(A*B).transpose() == B.transpose()*A.transpose()
```

```
True
```

By now, you can probably guess the matrix method for checking if a matrix is Hermitian.

```
A = matrix(QQbar, [[     45, -5-12*I, -1-15*I, -56-8*I],
                   [-5+12*I,      42,    32*I, -14-8*I],
                   [-1+15*I,   -32*I,      57,    12+I],
                   [-56+8*I, -14+8*I,    12-I,      93]])
A.is_hermitian()
```

```
True
```

We can illustrate the most fundamental property of a Hermitian matrix. The vectors `x` and `y` below are random, but according to Theorem HMIP the final command should produce `True` for any possible values of these two vectors. (You would be right to think that using random vectors over `QQbar` would be a better idea, but at this writing, these vectors are not as "random" as one would like, and are insufficient to perform an accurate test here.)

```
x = random_vector(QQ, 4) + QQbar(I)*random_vector(QQ, 4)
y = random_vector(QQ, 4) + QQbar(I)*random_vector(QQ, 4)
(A*x).hermitian_inner_product(y) == x.hermitian_inner_product(A*y)
```

```
True
```

Section MISLE: Matrix Inverses and Systems of Linear Equations

MISLE Matrix Inverse, Systems of Equations

We can use the computational method described in this section in hopes of finding a matrix inverse, as Theorem CINM gets us halfway there. We will continue with the matrix from Example MI. First we check that the matrix is nonsingular so we can apply the theorem, then we get "half" an inverse, and verify that it also behaves as a "full" inverse by meeting the full definition of a matrix inverse (Definition MI).

```
A = matrix(QQ, [[ 1,  2,  1,  2,  1],
                [-2, -3,  0, -5, -1],
                [ 1,  1,  0,  2,  1],
                [-2, -3, -1, -3, -2],
                [-1, -3, -1, -3,  1]])
A.is_singular()
```

```
False
```

```
I5 = identity_matrix(5)
M = A.augment(I5); M
```

```
[ 1  2  1  2  1  1  0  0  0  0]
[-2 -3  0 -5 -1  0  1  0  0  0]
[ 1  1  0  2  1  0  0  1  0  0]
[-2 -3 -1 -3 -2  0  0  0  1  0]
[-1 -3 -1 -3  1  0  0  0  0  1]
```

```
N = M.rref(); N
```

```
[ 1  0  0  0  0 -3  3  6 -1 -2]
[ 0  1  0  0  0  0 -2 -5 -1  1]
[ 0  0  1  0  0  1  2  4  1 -1]
[ 0  0  0  1  0  1  0  1  1  0]
[ 0  0  0  0  1  1 -1 -2  0  1]
```

```
J = N.matrix_from_columns(range(5,10)); J
```

```
[-3  3  6 -1 -2]
[ 0 -2 -5 -1  1]
[ 1  2  4  1 -1]
[ 1  0  1  1  0]
[ 1 -1 -2  0  1]
```

```
A*J == I5
```

```
True
```

```
J*A == I5
```

```
True
```

Note that the matrix `J` is constructed by taking the last 5 columns of `N` (numbered 5 through 9) and using them in the `matrix_from_columns()` matrix method. What happens if you apply the procedure above to a singular matrix? That would be an instructive experiment to conduct.

With an inverse of a coefficient matrix in hand, we can easily solve systems of equations, in the style of Example SABMI. We will recycle the matrices `A` and its inverse, `J`, from above, so be sure to run those compute cells first if you are playing along. We consider a system with `A` as a coefficient matrix and solve a linear system twice, once the old way and once the new way. Recall that with a nonsingular coefficient matrix, the solution will be unique for any choice of `const`, so you can experiment by changing the vector of constants and re-executing the code.

```
const = vector(QQ, [3, -4, 2, 1, 1])
A.solve_right(const)
```

```
(-12, -2, 3, 6, 4)
```

```
J*const
```

```
(-12, -2, 3, 6, 4)
```

```
A.solve_right(const) == J*const
```

```
True
```

Section MINM: Matrix Inverses and Nonsingular Matrices

MI Matrix Inverse

Now we know that invertibility is equivalent to nonsingularity, and that the procedure outlined in Theorem CINM will always yield an inverse for a nonsingular matrix. But rather than using that procedure, Sage implements a `.inverse()` method. In the following, we compute the inverse of a 3×3 matrix, and then purposely convert it to a singular matrix by replacing the last column by a linear combination of the first two.

```
A = matrix(QQ,[[ 3,  7, -6],
               [ 2,  5, -5],
               [-3, -8,  8]])
A.is_singular()
```

```
False
```

```
Ainv = A.inverse(); Ainv
```

```
[ 0  8  5]
[ 1 -6 -3]
[ 1 -3 -1]
```

```
A*Ainv
```

```
[1 0 0]
[0 1 0]
[0 0 1]
```

```
col_0 = A.column(0)
col_1 = A.column(1)
C = column_matrix([col_0, col_1, 2*col_0 - 4*col_1])
C.is_singular()
```

```
True
```

```
C.inverse()
```

```
Traceback (most recent call last):
...
ZeroDivisionError: input matrix must be nonsingular
```

Notice how the failure to invert `C` is explained by the matrix being singular.

Systems with nonsingular coefficient matrices can be solved easily with the matrix inverse. We will recycle `A` as a coefficient matrix, so be sure to execute the code above.

```
const = vector(QQ, [2, -1, 4])
A.solve_right(const)
```

```
(12, -4, 1)
```

```
A.inverse()*const
```

```
(12, -4, 1)
```

```
A.solve_right(const) == A.inverse()*const
```

```
True
```

If you find it more convenient, you can use the same notation as the text for a matrix inverse.

```
A^-1
```

```
[ 0  8  5]
[ 1 -6 -3]
[ 1 -3 -1]
```

NME3 Nonsingular Matrix Equivalences, Round 3

For square matrices, Sage has the methods `.is_singular()` and `.is_invertible()`. By Theorem NI we know these two functions to be logical opposites. One way to express this is that these two methods will always return different values. Here we demonstrate with a nonsingular matrix and a singular matrix. The comparison =! is "not equal."

```
nonsing = matrix(QQ, [[ 0, -1,  1, -3],
                      [ 1,  1,  0, -3],
                      [ 0,  4, -3,  8],
                      [-2, -4,  1,  5]])
nonsing.is_singular() != nonsing.is_invertible()
```

```
True
```

```
sing = matrix(QQ, [[ 1, -2, -6,  8],
                   [-1,  3,  7, -8],
                   [ 0, -4, -3, -2],
                   [ 0,  3,  1,  4]])
sing.is_singular() != sing.is_invertible()
```

```
True
```

We could test other properties of the matrix inverse, such as Theorem SS.

```
A = matrix(QQ, [[ 3, -5, -2,  8],
                [-1,  2,  0, -1],
                [-2,  4,  1, -4],
                [ 4, -5,  0,  8]])
B = matrix(QQ, [[ 1,  2,  4, -1],
                [-2, -3, -8, -2],
                [-2, -4, -7,  5],
                [ 2,  5,  7, -8]])
A.is_invertible() and B.is_invertible()
```

```
True
```

```
(A*B).inverse() == B.inverse()*A.inverse()
```

```
True
```

UM Unitary Matrices

No surprise about how we check if a matrix is unitary. Here is Example UM3,

```
A = matrix(QQbar, [
    [(1+I)/sqrt(5), (3+2*I)/sqrt(55), (2+2*I)/sqrt(22)],
    [(1-I)/sqrt(5), (2+2*I)/sqrt(55),  (-3+I)/sqrt(22)],
    [    I/sqrt(5), (3-5*I)/sqrt(55),    (-2)/sqrt(22)]
                ])
A.is_unitary()
```

```
True
```

```
A.conjugate_transpose() == A.inverse()
```

```
True
```

We can verify Theorem UMPIP, where the vectors `u` and `v` are created randomly. Try evaluating this compute cell with your own choices.

```
u = random_vector(QQ, 3) + QQbar(I)*random_vector(QQ, 3)
v = random_vector(QQ, 3) + QQbar(I)*random_vector(QQ, 3)
(A*u).hermitian_inner_product(A*v) == u.hermitian_inner_product(v)
```

```
True
```

If you want to experiment with permutation matrices, Sage has these too. We can create a permutation matrix from a list that indicates for each column the row with a one in it. Notice that the product here of two permutation matrices is again a permutation matrix.

```
sigma = Permutation([2,3,1])
S = sigma.to_matrix(); S
```

```
[0 0 1]
[1 0 0]
[0 1 0]
```

```
tau = Permutation([1,3,2])
T = tau.to_matrix(); T
```

```
[1 0 0]
[0 0 1]
[0 1 0]
```

```
S*T
```

```
[0 1 0]
[1 0 0]
[0 0 1]
```

```
rho = Permutation([2, 1, 3])
S*T == rho.to_matrix()
```

```
True
```

Section CRS: Column and Row Spaces

CSCS Column Space and Consistent Systems

We could compute the column space of a matrix with a span of the set of columns of the matrix, much as we did back in Sage CSS when we were checking consistency of linear systems using spans of the set of columns of a coefficent matrix. However, Sage provides a convenient matrix method to construct this same span: `.column_space()`. Here is a check.

```
D = matrix(QQ, [[ 2, -1, -4],
                [ 5,  2, -1],
                [-3,  1,  5]])
cs = D.column_space(); cs
```

```
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[    1     0 -11/9]
[    0     1  -1/9]
```

```
cs_span = (QQ^3).span(D.columns())
cs == cs_span
```

```
True
```

In Sage CSS we discussed solutions to systems of equations and the membership of the vector of constants in the span of the columns of the coefficient matrix. This discussion turned on Theorem SLSLC. We can now be slightly more efficient with Theorem CSCS, while still using the same ideas. We recycle the computations from Example CSMCS that use Archetype D and Archetype E.

```
coeff = matrix(QQ, [[ 2, 1,  7, -7],
                    [-3, 4, -5, -6],
                    [ 1, 1,  4, -5]])
constD = vector(QQ, [8, -12, 4])
constE = vector(QQ, [2, 3, 2])
cs = coeff.column_space()
coeff.solve_right(constD)
```

```
(4, 0, 0, 0)
```

```
constD in cs
```

```
True
```

```
coeff.solve_right(constE)
```

```
Traceback (most recent call last):
...
ValueError: matrix equation has no solutions
```

```
constE in cs
```

```
False
```

CSOC Column Space, Original Columns

It might be worthwhile for Sage to create a column space using actual columns of the matrix as a spanning set. But we can do it ourselves fairly easily. A discussion follows the example.

```
A = matrix(QQ, [[-1, -1, 0,  1,  0,  1, -2, -3, 0],
                [-1,  0, 0,  5, -2,  6, -6,  3, 5],
                [ 0,  0, 1, -6, -1,  0, -5,  0, 5],
                [ 2,  2, 1, -8, -2,  0, -4,  8, 8],
                [ 2,  2, 0, -2, -1,  0,  1,  8, 3],
                [ 2,  1, 0, -6, -1, -1, -1,  6, 4]])
A.rref()
```

```
[ 1  0  0 -5  0 -2  0  1  1]
[ 0  1  0  4  0  1  2  2 -1]
[ 0  0  1 -6  0 -2 -2 -2  2]
[ 0  0  0  0  1 -2  3 -2 -3]
[ 0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0]
```

```
B = A.matrix_from_columns(A.pivots())
A.column_space() == B.column_space()
```

```
True
```

```
B.rref()
```

```
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
[0 0 0 0]
[0 0 0 0]
```

```
cs = (QQ^6).span_of_basis(B.columns())
cs
```

```
Vector space of degree 6 and dimension 4 over Rational Field
User basis matrix:
[-1 -1  0  2  2  2]
[-1  0  0  2  2  1]
[ 0  0  1  1  0  0]
[ 0 -2 -1 -2 -1 -1]
```

```
cs.basis()
```

```
[
(-1, -1, 0, 2, 2, 2),
(-1, 0, 0, 2, 2, 1),
(0, 0, 1, 1, 0, 0),
(0, -2, -1, -2, -1, -1)
]
```

```
cs.echelonized_basis()
```

```
[
(1, 0, 0, 0, 0, 5),
(0, 1, 0, 0, 0, -1),
(0, 0, 1, 0, -1, -3),
(0, 0, 0, 1, 1, 3)
]
```

```
cs == A.column_space()
```

```
True
```

```
cs2 = (QQ^6).span_of_basis([A.column(i) for i in A.pivots()])
cs2 == A.column_space()
```

```
True
```

We see that `A` has four pivot columns, numbered `0,1,2,4`. The matrix `B` is just a convenience to hold the pivot columns of `A`. However, the column spaces of `A` and `B` should be equal, as Sage verifies. Also `B` will row-reduce to the same 0-1 pivot columns of the reduced row-echelon form of the full matrix `A`. So it is no accident that the reduced row-echelon form of `B` is a full identity matrix, followed by sufficiently many zero rows to give the matrix the correct size.

The vector space method `.span_of_basis()` is new to us. It creates a span of a set of vectors, as before, but we now are responsible for supplying a linearly independent set of vectors. Which we have done. We know this because Theorem BCS guarantees the set we provided is linearly independent (and spans the column space), while Sage would have given us an error if we had provided a linearly dependent set. In return, Sage will carry this linearly independent spanning set along with the vector space, something Sage calls a "user basis."

Notice how `cs` has two linearly independent spanning sets now. Our set of "original columns" is obtained via the standard vector space method `.basis()` and we can obtain a linearly independent spanning set that looks more familiar with the vector space method `.echelonized_basis()`. For a vector space created with a simple `.span()` construction these two commands would yield identical results — it is only when we supply a linearly independent spanning set with the `.span_of_basis()` method that a "user basis" becomes relevant. Finally, we check that `cs` is indeed the column space of `A` (we knew it would be) and then we provide a one-line, totally general construction of the column space using original columns. This is an opportunity to make an interesting observation, which could be used to substantiate several theorems. When we take the original columns that we recognize as pivot columns, and use them alone to form a matrix, this new matrix *will always* row-reduce to an identity matrix followed by zero rows. This is basically a consequence of reduced row-echelon form. Evaluate the compute cell below repeatedly. The number of columns could in theory change, though this is unlikely since the columns of a random matrix are unlikely to be linearly dependent. In any event, the form of the result will always be an identity matrix followed by some zero rows.

```
F = random_matrix(QQ, 5, 3)
F.matrix_from_columns(F.pivots()).rref()   # random
```

```
[1 0 0]
[0 1 0]
[0 0 1]
[0 0 0]
[0 0 0]
```

With more columns than rows, we know by Theorem MVSLD that we will have a reduced number of pivot columns. Here, we will almost always see an identity matrix as the result, though we could get a smaller identity matrix followed by zero rows.

```
F = random_matrix(QQ, 3, 5)
F.matrix_from_columns(F.pivots()).rref()   # random
```

```
[1 0 0]
[0 1 0]
[0 0 1]
```

NME4 Nonsingular Matrices, Round 4

Archetype A and Archetype B have square coefficient matrices that illustrate the dichotomy of singular and nonsingular matrices. Here we illustrate the latest addition to our series of equivalences, Theorem CSNM.

```
A = matrix(QQ, [[1, -1, 2],
                [2,  1, 1],
                [1,  1, 0]])
B = matrix(QQ, [[-7, -6, -12],
                [ 5,  5,   7],
                [ 1,  0,   4]])
A.is_singular()
```

```
True
```

```
A.column_space() == QQ^3
```

```
False
```

```
B.is_singular()
```

```
False
```

```
B.column_space() == QQ^3
```

```
True
```

We can even write Theorem CSNM as a one-line Sage statement that will return `True` for *any* square matrix. Run the following repeatedly and it should always return `True`. We have kept the size of the matrix relatively small to be sure that some of the random matrices produced are singular.

```
A = random_matrix(QQ, 4, 4)
A.is_singular() == (not A.column_space() == QQ^4)
```

```
True
```

RSM Row Space of a Matrix

Not to be outdone, and not suprisingly, Sage can compute a row space with the matrix method `.row_space()`. Indeed, given Sage's penchant for treating vectors as rows, much of Sage's infrastructure for vector spaces ultimately relies on Theorem REMRS. More on that in Sage SUTH0. For now, we reprise Example IAS as an illustration.

```
v1 = vector(QQ, [1,  2,  1,  6,   6])
v2 = vector(QQ, [3, -1,  2, -1,   6])
v3 = vector(QQ, [1, -1,  0, -1,  -2])
v4 = vector(QQ, [-3, 2, -3,  6, -10])
X = (QQ^5).span([v1, v2, v3, v4])
```

```
A = matrix([v1, v2, v3, v4])
rsA = A.row_space()
X == rsA
```

```
True
```

```
B = A.rref()
rsB = B.row_space()
X == rsB
```

```
True
```

```
X
```

```
Vector space of degree 5 and dimension 3 over Rational Field
Basis matrix:
[ 1  0  0  2 -1]
[ 0  1  0  3  1]
[ 0  0  1 -2  5]
```

```
X.basis()
```

```
[
(1, 0, 0, 2, -1),
(0, 1, 0, 3, 1),
(0, 0, 1, -2, 5)
]
```

```
B
```

```
[ 1  0  0  2 -1]
[ 0  1  0  3  1]
[ 0  0  1 -2  5]
[ 0  0  0  0  0]
```

We begin with the same four vectors in Example IAS and create their span, the vector space `X`. The matrix `A` has these four vectors as rows and `B` is the reduced row-echelon form of `A`. Then the row spaces of `A` and `B` are equal to the vector space `X` (and each other). The way Sage describes this vector space is with a matrix whose rows *are the nonzero rows of the reduced row-echelon form of the matrix* `A`. This is Theorem BRS in action where we go with Sage's penchant for rows and ignore the text's penchant for columns.

We can illustrate a few other results about row spaces with Sage. Discussion follows.

```
A = matrix(QQ,[[1,  1, 0, -1,  1,  1, 0],
               [4,  5, 1, -6,  1,  6, 1],
               [5,  5, 1, -5,  4,  5, 2],
               [3, -1, 0,  5, 11, -5, 4]])
A.row_space() == A.transpose().column_space()
```

```
True
```

```
B = matrix(QQ,[[ 7,  9, 2, -11,   1, 11,  2],
               [-4, -3, 1,   2,  -7, -2,  1],
               [16,  8, 2,   0,  30,  0, 12],
               [ 2, 10, 2, -18, -16, 18, -4]])
B.column_space() == B.transpose().row_space()
```

```
True
```

```
A.rref() == B.rref()
```

```
True
```

```
A.row_space() == B.row_space()
```

```
True
```

We use the matrix `A` to illustrate Definition RSM, and the matrix `B` to illustrate Theorem CSRST. `A` and `B` were designed to have the same reduced row-echelon form, and hence be row-equivalent, so this is not a consequence of any theorem or previous computation. However, then Theorem REMRS guarantees that the row spaces of `A` and `B` are equal.

Section FS: Four Subsets

LNS Left Null Spaces

Similar to (right) null spaces, a left null space can be computed in Sage with the matrix method `.left_kernel()`. For a matrix A, elements of the (right) null space are vectors $\mathbf{x}$ such that $A\mathbf{x} = \mathbf{0}$. If we interpret a vector placed to the left of a matrix as a 1-row matrix, then the product $\mathbf{x}A$ can be interpreted, according to our definition of matrix multiplication (Definition MM), as the entries of the vector $\mathbf{x}$ providing the scalars for a linear combination of the *rows* of the matrix A. So you can view each vector in the left null space naturally as the entries of a matrix with a single row, Y, with the property that $YA = \mathcal{O}$.

Given Sage's preference for row vectors, the simpler name `.kernel()` is a synonym for `.left_kernel()`. Given your text's preference for column vectors, we will continue to rely on the `.right_kernel()`. Left kernels in Sage have the same options as right kernels and produce vector spaces as output in the same manner. Once created, a vector space all by itself (with no history) is neither left or right. Here is a quick repeat of Example LNS.

```
A = matrix(QQ, [[ 1, -3, 1],
                [-2,  1, 1],
                [ 1,  5, 1],
                [ 9, -4, 0]])
A.left_kernel(basis='pivot')
```

```
Vector space of degree 4 and dimension 1 over Rational Field
User basis matrix:
[-2  3 -1  1]
```

RRSM Row-Reducing a Symbolic Matrix

Sage can very nearly reproduce the reduced row-echelon form we obtained from the augmented matrix with variables present in the final column. The first line below is a bit of advanced Sage. It creates a number system `R` mixing the rational numbers with the variables `b1` through `b6`. It is not important to know the details of this construction right now. `B` is the reduced row-echelon form of `A`, as computed by Sage, where we have displayed the last column separately so it will all fit. You will notice that `B` is different than what we used in Example CSANS, where all the differences are in the final column.

However, we can perform row operations on the final two rows of `B` to bring Sage's result in line with what we used above for the final two entries of the last column, which are the most critical. Notice that since the final two rows are almost all zeros, any sequence of row operations on just these two rows will preserve the zeros (and we need only display the final column to keep track of our progress).

```
R.<b1,b2,b3,b4,b5,b6> = QQ[]
A = matrix(R, [[ 10,  0,  3,   8,   7, b1],
               [-16, -1, -4, -10, -13, b2],
               [ -6,  1, -3,  -6,  -6, b3],
               [  0,  2, -2,  -3,  -2, b4],
               [  3,  0,  1,   2,   3, b5],
               [ -1, -1,  1,   1,   0, b6]])
A
```

```
[ 10   0   3   8   7  b1]
[-16  -1  -4 -10 -13  b2]
[ -6   1  -3  -6  -6  b3]
[  0   2  -2  -3  -2  b4]
[  3   0   1   2   3  b5]
[ -1  -1   1   1   0  b6]
```

```
B = copy(A.echelon_form())
B[0:6,0:5]
```

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

```
B[0:6,5]
```

```
[ -1/4*b1 - 5/4*b2 + 11/4*b3 - 2*b4]
[                 3*b2 - 8*b3 + 6*b4]
[ -3/2*b1 + 3/2*b2 - 13/2*b3 + 4*b4]
[                  b1 + b2 - b3 + b4]
[     1/4*b1 + 1/4*b2 + 1/4*b3 + b5]
[1/4*b1 - 3/4*b2 + 9/4*b3 - b4 + b6]
```

```
B.rescale_row(4, 4); B[0:6, 5]
```

```
[ -1/4*b1 - 5/4*b2 + 11/4*b3 - 2*b4]
[                3*b2 - 8*b3 + 6*b4]
[ -3/2*b1 + 3/2*b2 - 13/2*b3 + 4*b4]
[                 b1 + b2 - b3 + b4]
[               b1 + b2 + b3 + 4*b5]
[1/4*b1 - 3/4*b2 + 9/4*b3 - b4 + b6]
```

```
B.add_multiple_of_row(5, 4, -1/4); B[0:6, 5]
```

```
[-1/4*b1 - 5/4*b2 + 11/4*b3 - 2*b4]
[               3*b2 - 8*b3 + 6*b4]
[-3/2*b1 + 3/2*b2 - 13/2*b3 + 4*b4]
[                b1 + b2 - b3 + b4]
[              b1 + b2 + b3 + 4*b5]
[        -b2 + 2*b3 - b4 - b5 + b6]
```

```
B.rescale_row(5, -1); B[0:6, 5]
```

```
[-1/4*b1 - 5/4*b2 + 11/4*b3 - 2*b4]
[               3*b2 - 8*b3 + 6*b4]
[-3/2*b1 + 3/2*b2 - 13/2*b3 + 4*b4]
[                b1 + b2 - b3 + b4]
[              b1 + b2 + b3 + 4*b5]
[         b2 - 2*b3 + b4 + b5 - b6]
```

```
B.add_multiple_of_row(4, 5,-1); B[0:6, 5]
```

```
[-1/4*b1 - 5/4*b2 + 11/4*b3 - 2*b4]
[               3*b2 - 8*b3 + 6*b4]
[-3/2*b1 + 3/2*b2 - 13/2*b3 + 4*b4]
[                b1 + b2 - b3 + b4]
[       b1 + 3*b3 - b4 + 3*b5 + b6]
[         b2 - 2*b3 + b4 + b5 - b6]
```

Notice that the last two entries of the final column now have just a single `b1` and a single `B2`. We could continue to perform more row operations by hand, using the last two rows to progressively eliminate `b1` and `b2` from the other four expressions of the last column. Since the two last rows have zeros in their first five entries, only the entries in the final column would change. You will see that much of this section is about how to automate these final calculations.

EEF Extended Echelon Form

Sage will compute the extended echelon form, an improvement over what we did "by hand" in Sage MISLE. And the output can be requested as a "subdivided" matrix so that we can easily work with the pieces C and L. Pieces of subdivided matrices can be extracted with indices entirely similar to how we index the actual entries of a matrix. We will redo Example FS2 as an illustration of Theorem FS and Theorem PEEF.

```
A = matrix(QQ,[[ 10,  0,  3,   8,   7],
               [-16, -1, -4, -10, -13],
               [ -6,  1, -3,  -6,  -6],
               [  0,  2, -2,  -3,  -2],
               [  3,  0,  1,   2,   3],
               [ -1, -1,  1,   1,   0]])
N = A.extended_echelon_form(subdivide=True); N
```

```
[ 1  0  0  0  2| 0  0  1 -1  2 -1]
[ 0  1  0  0 -3| 0  0 -2  3 -3  3]
[ 0  0  1  0  1| 0  0  1  1  3  3]
[ 0  0  0  1 -2| 0  0 -2  1 -4  0]
[--------------+-----------------]
[ 0  0  0  0  0| 1  0  3 -1  3  1]
[ 0  0  0  0  0| 0  1 -2  1  1 -1]
```

```
C = N.subdivision(0,0); C
```

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

```
L = N.subdivision(1,1); L
```

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

```
K = N.subdivision(0,1); K
```

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

```
C.rref() == C
```

```
True
```

```
L.rref() == L
```

```
True
```

```
A.right_kernel() == C.right_kernel()
```

```
True
```

```
A.row_space() == C.row_space()
```

```
True
```

```
A.column_space() == L.right_kernel()
```

```
True
```

```
A.left_kernel() == L.row_space()
```

```
True
```

```
J = N.matrix_from_columns(range(5, 11)); J
```

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

```
J.is_singular()
```

```
False
```

```
J*A
```

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

```
J*A == A.rref()
```

```
True
```

Notice how we can employ the uniqueness of reduced row-echelon form (Theorem RREFU) to verify that `C` and `L` are in reduced row-echelon form. Realize too, that subdivided output is an optional behavior of the `.extended_echelon_form()` method and must be requested with `subdivide=True`.

Additionally, it is the uniquely determined matrix `J` that provides us with a standard version of the final column of the row-reduced augmented matrix using variables in the vector of constants, as first shown back in Example CSANS and verified here with variables defined as in Sage RRSM. (The vector method `.column()` converts a vector into a 1-column matrix, used here to format the matrix-vector product nicely.)

```
R.<b1,b2,b3,b4,b5,b6> = QQ[]
const = vector(R, [b1, b2, b3, b4, b5, b6])
(J*const).column()
```

```
[       b3 - b4 + 2*b5 - b6]
[-2*b3 + 3*b4 - 3*b5 + 3*b6]
[     b3 + b4 + 3*b5 + 3*b6]
[         -2*b3 + b4 - 4*b5]
[b1 + 3*b3 - b4 + 3*b5 + b6]
[  b2 - 2*b3 + b4 + b5 - b6]
```

Chapter VS
Vector Spaces

Section S: Subspaces

VS Vector Spaces

Our conception of a vector space has become much broader with the introduction of abstract vector spaces — those whose elements ("vectors") are not just column vectors, but polynomials, matrices, sequences, functions, etc. Sage is able to perform computations using many different abstract and advanced ideas (such as derivatives of functions), but in the case of linear algebra, Sage will primarily stay with vector spaces of column vectors. Chapter R, and specifically, Section VR and Sage SUTH2 will show us that this is not as much of a limitation as it might first appear.

While limited to vector spaces of column vectors, Sage has an impressive range of capabilities for vector spaces, which we will detail throughout this chapter. You may have already noticed that many questions about abstract vector spaces can be translated into questions about column vectors. This theme will continue, and Sage commands we already know will often be helpful in answering these questions.

Theorem SSS, Theorem NSMS, Theorem CSMS, Theorem RSMS and Theorem LNSMS each tells us that a certain set is a subspace. The first is the abstract version of creating a subspace via the span of a set of vectors, but still applies to column vectors as a special case. The remaining four all begin with a matrix and create a subspace of column vectors. We have created these spaces many times already, but notice now that the description Sage outputs explicitly says they are vector spaces, and that there are still some parts of the output that we need to explain. Here are two reminders, first a span, and then a vector space created from a matrix.

```
V = QQ^4
v1 = vector(QQ, [ 1, -1, 2, 4])
v2 = vector(QQ, [-3,  0, 2, 1])
v3 = vector(QQ, [-1, -2, 6, 9])
W = V.span([v1, v2, v3])
W
```

```
Vector space of degree 4 and dimension 2 over Rational Field
Basis matrix:
[    1     0  -2/3  -1/3]
[    0     1  -8/3 -13/3]
```

```
A = matrix(QQ, [[1, 2, -4,  0, -4],
                [0, 1, -1, -1, -1],
                [3, 2, -8,  4, -8]])
```

```
W = A.column_space()
W
```

```
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1  0  3]
[ 0  1 -4]
```

Section B: Bases

B Bases

Every vector space in Sage has a basis — you can obtain this with the vector space method `.basis()`, and the result is a list of vectors. Another method for a vector space is `.basis_matrix()` which outputs a matrix whose rows are the vectors of a basis. Sometimes one form is more convenient than the other, but notice that the description of a vector space chooses to print the basis matrix (since its display is just a bit easier to read). A vector space typically has many bases (infinitely many), so which one does Sage use? You will notice that the basis matrices displayed are in reduced row-echelon form — this is the defining property of the basis chosen by Sage.

Here is Example RSB again as an example of how bases are provided in Sage.

```
V = QQ^3
v1 = vector(QQ, [ 2, -3,  1])
v2 = vector(QQ, [ 1,  4,  1])
v3 = vector(QQ, [ 7, -5,  4])
v4 = vector(QQ, [-7, -6, -5])
W = V.span([v1, v2, v3, v4])
W
```

```
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[   1    0 7/11]
[   0    1 1/11]
```

```
W.basis()
```

```
[
(1, 0, 7/11),
(0, 1, 1/11)
]
```

```
W.basis_matrix()
```

```
[   1    0 7/11]
[   0    1 1/11]
```

SUTH0 Sage Under The Hood, Round 0

Or perhaps, "under the bonnet" if you learned your English in the Commonwealth. This is the first in a series that aims to explain how our knowledge of linear algebra *theory* helps us understand the design, construction and informed use of Sage.

How does Sage determine if two vector spaces are equal? Especially since these are infinite sets? One approach would be to take a spanning set for the first vector space (maybe a minimal spanning set), and ask if each element of the spanning set is an element of the second vector space. If so, the first vector space is a subset of the second. Then we could turn it around, and determine if the second vector space is a subset of the first. By Definition SE, the two vector spaces would be equal if both subset tests succeeded.

However, each time we would test if an element of a spanning set lives in a second vector space, we would need to solve a linear system. So for two large vector spaces, this could take a noticeable amount of time. There is a better way, made possible by exploiting two important theorems.

For every vector space, Sage creates a basis that uniquely identifies the vector space. We could call this a "canonical basis." By Theorem REMRS we can span the row space of a matrix by the rows of any row-equivalent matrix. So if we begin with a vector space described by any basis (or any spanning set, for that matter), we can make a matrix with these rows as vectors, and the vector space is now the row space of the matrix. Of all the possible row-equivalent matrices, which would you pick? Of course, the reduced row-echelon version is useful, and here it is critical to realize this version is unique (Theorem RREFU).

So for every vector space, Sage takes a spanning set, makes its vectors the rows of a matrix, row-reduces the matrix and tosses out the zero rows. The result is what Sage calls an "echelonized basis." Now, two vector spaces are equal if, and only if, they have equal "echelonized basis matrices." It takes some computation to form the echelonized basis, but once built, the comparison of two echelonized bases can proceed very quickly by perhaps just comparing entries of the echelonized basis matrices.

You might create a vector space with a basis you prefer (a "user basis"), but Sage always has an echelonized basis at hand. If you do not specify some alternate basis, this is the basis Sage will create and provide for you. We can now continue a discussion we began back in Sage SSNS. We have consistently used the `basis='pivot'` keyword when we construct null spaces. This is because we initially prefer to see the basis described in Theorem BNS, rather than Sage's default basis, the echelonized version. But the echelonized version is always present and available.

```
A = matrix(QQ, [[14, -42, -2, -44, -42, 100, -18],
                [-40, 120, -6, 129, 135, -304, 28],
                [11, -33, 0, -35, -35, 81, -11],
                [-21, 63, -4, 68, 72, -161, 13],
                [-4, 12, -1, 13, 14, -31, 2]])
K = A.right_kernel(basis='pivot')
K.basis_matrix()
```

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

```
K.echelonized_basis_matrix()
```

```
[ 1  0  0  0 -4 -2 -1]
[ 0  1  0  0 12  6  3]
[ 0  0  1  0 -2 -1 -1]
```

```
[ 0  0  0  1 -3 -1 -1]
```

NME5 Nonsingular Matrix Equivalences, Round 5

We can easily illustrate our latest equivalence for nonsingular matrices.

```
A = matrix(QQ, [[ 2,  3, -3,  2,  8, -4],
                [ 3,  4, -4,  4,  8,  1],
                [-2, -2,  3, -3, -2, -7],
                [ 0,  1, -1,  2,  3,  4],
                [ 2,  1,  0,  1, -4,  4],
                [ 1,  2, -2,  1,  7, -5]])
not A.is_singular()
```

```
True
```

```
V = QQ^6
cols = A.columns()
V == V.span(cols)
```

```
True
```

```
V.linear_dependence(cols) == []
```

```
True
```

C Coordinates

For vector spaces of column vectors, Sage can quickly determine the coordinates of a vector relative to a basis, as guaranteed by Theorem VRRB. We illustrate some new Sage commands with a simple example and then apply them to orthonormal bases. The vectors `v1` and `v2` are linearly independent and thus span a subspace with a basis of size 2. We first create this subspace and let Sage determine the basis, then we illustrate a new vector space method, `.subspace_with_basis()`, that allows us to specify the basis. (This method is very similar to `.span_of_basis()`, except it preserves a subspace relationship with the original vector space.) Notice how the description of the vector space makes it clear that `W` has a user-specified basis. Notice too that the actual subspace created is the same in both cases.

```
V = QQ^3
v1 = vector(QQ,[ 2, 1, 3])
v2 = vector(QQ,[-1, 1, 4])
U=V.span([v1,v2])
U
```

```
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[   1    0 -1/3]
[   0    1 11/3]
```

```
W = V.subspace_with_basis([v1, v2])
W
```

```
Vector space of degree 3 and dimension 2 over Rational Field
User basis matrix:
[ 2  1  3]
[-1  1  4]
```

```
U == W
```

```
True
```

Now we manufacture a third vector in the subspace, and request a coordinatization in each vector space, which has the effect of using a different basis in each case. The vector space method `.coordinate_vector(v)` computes a vector whose entries express `v` as a linear combination of basis vectors. Verify for yourself in each case below that the components of the vector returned really give a linear combination of the basis vectors that equals `v3`.

```
v3 = 4*v1 + v2; v3
```

```
(7, 5, 16)
```

```
U.coordinate_vector(v3)
```

```
(7, 5)
```

```
W.coordinate_vector(v3)
```

```
(4, 1)
```

Now we can construct a more complicated example using an orthonormal basis, specifically the one from Example CROB4, but we will compute over `QQbar`, the field of algebraic numbers. We form the four vectors of the orthonormal basis, install them as the basis of a vector space and then ask for the coordinates. Sage treats the square roots in the scalars as "symbolic" expressions, so we need to explicitly coerce them into `QQbar` before computing the scalar multiples.

```
V = QQbar^4
x1 = vector(QQbar, [    1+I,       1,      1-I,       I])
x2 = vector(QQbar, [  1+5*I,   6+5*I,     -7-I,   1-6*I])
x3 = vector(QQbar, [-7+34*I, -8-23*I, -10+22*I, 30+13*I])
x4 = vector(QQbar, [ -2-4*I,     6+I,    4+3*I,     6-I])
v1 = QQbar(1/sqrt(6))   * x1
v2 = QQbar(1/sqrt(174)) * x2
v3 = QQbar(1/sqrt(3451))* x3
v4 = QQbar(1/sqrt(119)) * x4
W = V.subspace_with_basis([v1, v2, v3, v4])
w = vector(QQbar, [2, -3, 1, 4])
c = W.coordinate_vector(w); c
```

```
(0.?e-14 - 2.0412414523194?*I,
-1.4403862828000? + 2.27429413073671?*I,
2.0427196489446? - 3.59178204939423?*I,
0.55001909821693? + 1.10003819643385?*I)
```

Is this right? Our exact coordinates in the text are printed differently, but we can check that they are the same numbers:

```
c[0] == 1/sqrt(6)*(-5*I)
```

```
True
```

```
c[1] == 1/sqrt(174)*(-19+30*I)
```

```
True
```

```
c[2] == 1/sqrt(3451)*(120-211*I)
```

```
True
```

```
c[3] == 1/sqrt(119)*(6+12*I)
```

```
True
```

With an orthonormal basis, we can illustrate Theorem CUMOS by making the four vectors the columns of 4×4 matrix and verifying the result is a unitary matrix.

```
U = column_matrix([v1, v2, v3, v4])
U.is_unitary()
```

```
True
```

We will see coordinate vectors again, in a more formal setting, in Sage VR.

Section D: Dimension

D Dimension

Now we recognize that every basis has the same size, even if there are many different bases for a given vector space. The dimension is an important piece of information about a vector space, so Sage routinely provides this as part of the description of a vector space. But it can be returned by itself with the vector space method `.dimension()`. Here is an example of a subspace with dimension 2.

```
V = QQ^4
v1 = vector(QQ, [2, -1, 3,  1])
v2 = vector(QQ, [3, -3, 4,  0])
v3 = vector(QQ, [1, -2, 1, -1])
v4 = vector(QQ, [4, -5, 5, -1])
W = span([v1, v2, v3, v4])
W
```

```
Vector space of degree 4 and dimension 2 over Rational Field
Basis matrix:
[  1   0 5/3   1]
[  0   1 1/3   1]
```

```
W.dimension()
```

```
2
```

RNM Rank and Nullity of a Matrix

The rank and nullity of a matrix in Sage could be exactly what you would have guessed. But we need to be careful. The rank is the rank. But nullity in Sage is the dimension of the *left* nullspace. So we have matrix methods `.nullity()`, `.left_nullity()`, `.right_nullity()`, where the first two are equal and correspond to Sage's preference for rows, and the third is the column version used by the text. That said, a "row version" of Theorem RPNC is also true.

```
A = matrix(QQ, [[-1, 0, -4, -3,  1, -1,  0,  1, -1],
                [ 1, 1,  6,  6,  5,  3,  4, -5,  3],
                [ 2, 0,  7,  5, -3,  1, -1, -1,  2],
                [ 2, 1,  6,  6,  3,  1,  3, -3,  5],
                [-2, 0, -1, -1,  3,  3,  1, -3, -4],
                [-1, 1,  4,  4,  7,  5,  4, -7, -1]])
A.rank()
```

```
4
```

```
A.right_nullity()
```

```
5
```

```
A.rank() + A.right_nullity() == A.ncols()
```

```
True
```

```
A.rank() + A.left_nullity() == A.nrows()
```

```
True
```

NME6 Nonsingular Matrix Equivalences, Round 6

Recycling the nonsingular matrix from Sage NME5 we can use Sage to verify the two new equivalences of Theorem NME6.

```
A = matrix(QQ, [[ 2,  3, -3,  2,  8, -4],
                [ 3,  4, -4,  4,  8,  1],
                [-2, -2,  3, -3, -2, -7],
                [ 0,  1, -1,  2,  3,  4],
                [ 2,  1,  0,  1, -4,  4],
                [ 1,  2, -2,  1,  7, -5]])
not A.is_singular()
```

```
True
```

```
A.rank() == A.ncols()
```

```
True
```

```
A.right_nullity() == 0
```

```
True
```

Section PD: Properties of Dimension

DMS Dimensions of Matrix Subspaces

The theorems in this section about the dimensions of various subspaces associated with matrices can be tested easily in Sage. For a large arbitrary matrix, we first verify Theorem RMRT, followed by the four conclusions of Theorem DFS.

```
A = matrix(QQ, [
    [ 1, -2,  3,  2,  0,  2,  2, -1,  3,  8,  0,  7],
    [-1,  2, -2, -1,  3, -3,  5, -2, -6, -7,  6, -2],
    [ 0,  0,  1,  1,  0,  0,  1, -3, -2,  0,  3,  8],
    [-1, -1,  0, -1, -1,  0, -6, -2, -5, -6,  5,  1],
    [ 1, -3,  2,  1, -4,  4, -6,  2,  7,  7, -5,  2],
    [-2,  2, -2, -2,  3, -3,  6, -1, -8, -8,  7, -7],
    [ 0, -3,  2,  0, -3,  3, -7,  1,  2,  3, -1,  0],
    [ 0, -1,  2,  1,  2,  0,  4, -3, -3,  2,  6,  6],
    [-1,  1,  0, -1,  2, -1,  6, -2, -6, -3,  8,  0],
    [ 0, -4,  4,  0, -2,  4, -4, -2, -2,  4,  8,  6]
             ])
m = A.nrows()
n = A.ncols()
r = A.rank()
m, n, r
```

```
(10, 12, 7)
```

```
A.transpose().rank() == r
```

```
True
```

```
A.right_kernel().dimension() == n - r
```

```
True
```

```
A.column_space().dimension() == r
```

```
True
```

```
A.row_space().dimension() == r
```

```
True
```

```
A.left_kernel().dimension() == m - r
```

```
True
```

Chapter D
Determinants

Section DM: Determinant of a Matrix

EM Elementary Matrices

Each of the three types of elementary matrices can be constructed easily, with syntax similar to methods for performing row operations on matrices. Here we have three 4×4 elementary matrices, in order: $E_{1,3}$, $E_2(7)$, $E_{2,4}(9)$. Notice the change in numbering on the rows, and the order of the parameters.

```
A = elementary_matrix(QQ, 4, row1=0, row2=2); A
```

```
[0 0 1 0]
[0 1 0 0]
[1 0 0 0]
[0 0 0 1]
```

```
A = elementary_matrix(QQ, 4, row1=1, scale=7); A
```

```
[1 0 0 0]
[0 7 0 0]
[0 0 1 0]
[0 0 0 1]
```

```
A = elementary_matrix(QQ, 4, row1=3, row2=1, scale=9); A
```

```
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 9 0 1]
```

Notice that `row1` is always the row that is being changed. The keywords can be removed, but the `scale` keyword must be used to create the second type of elementary matrix, to avoid confusion with the first type.

We can illustrate some of the results of this section with two examples. First, we convert a matrix into a second matrix which is row-equivalent, and then accomplish the same thing with matrix multiplication and a product of elementary matrices.

```
A = matrix(QQ, [[6, -2, 3, -2],
                [3,  3, 1,  8],
                [4,  0, 5,  4]])
B = copy(A)
B.swap_rows(0,2)
E1 = elementary_matrix(QQ, 3, row1=0, row2=2)
B.rescale_row(1, 5)
E2 = elementary_matrix(QQ, 3, row1=1, scale=5)
B.add_multiple_of_row(1, 0, -3)
E3 = elementary_matrix(QQ, 3, row1=1, row2=0, scale=-3)
B
```

```
[  4   0   5   4]
[  3  15 -10  28]
[  6  -2   3  -2]
```

```
E3*E2*E1*A
```

```
[  4   0   5   4]
[  3  15 -10  28]
[  6  -2   3  -2]
```

```
B == E3*E2*E1*A
```

```
True
```

```
R = E3*E2*E1
R.is_singular()
```

```
False
```

```
R
```

```
[ 0  0  1]
[ 0  5 -3]
[ 1  0  0]
```

The matrix `R`, the product of three elementary matrices, can be construed as the collective effect of the three row operations employed. With more row operations, `R` might look even less like an identity matrix. As the product of nonsingular matrices (Theorem EMN), `R` is nonsingular (Theorem NPNT).

The matrix `B` above is not in reduced row-echelon form (it was just row-equivalent to `A`). What if we were to begin with a matrix and track all of the row operations required to bring the matrix to reduced row-echelon form? As above, we could form the associated elementary matrices and form their product, creating a single matrix `R` that captures all of the row operations.

It turns out we have already done this. Extended echelon form is the subject of Theorem PEEF, whose second conclusion says that $B = JA$, where A is the original matrix, and B is the row-equivalent matrix in reduced row-echelon form. Then J is a square nonsingular matrix that is the product of the sequence of elementary matrices associated with the sequence of row operations converting A into B. There may be many, many different sequences of row operations that convert A to B, but the requirement that extended echelon form be in reduced row-echelon form guarantees that J is unique.

DM Determinant of a Matrix

Computing the determinant in Sage is straightforward.

```
A = matrix(QQ,[[ 4,  3, -2, -9, -11, -14, -4,  11,  -4,   4],
               [ 0,  1,  0, -1,  -1,  -3, -2,   3,   6,  15],
               [ 0,  1,  1,  1,   3,   0, -3,  -5,  -9,  -3],
               [-3,  3,  3,  6,  12,   3, -9, -12,  -9,  15],
               [-2,  0,  2,  3,   5,   3, -3,  -5,  -3,   7],
               [ 3,  3, -1, -7,  -8, -12, -5,   8,  -4,   9],
               [ 0, -1,  1,  2,   2,   5,  3,  -6,  -8, -12],
               [-1, -2, -1,  1,  -2,   5,  6,   3,   6,  -8],
               [ 2,  1, -3, -6,  -8,  -7,  0,  12,  11,   8],
               [-3, -2,  2,  5,   5,   9,  2,  -7,  -4,  -3]])
A.determinant()
```

```
3
```

Random matrices, even with small entries, can have very large determinants.

```
B = matrix(QQ,  15, [
  [-5, -5, -5,  4,  1,  1, -3,  0,  4,  4, -2, -4,  2,  3, -1],
  [ 1,  1, -4,  3,  3,  4,  1, -1, -5,  4, -3,  0, -1,  0,  0],
  [ 3,  4, -2,  3, -1, -5, -1, -4, -5,  0, -1,  2, -4, -1, -1],
  [ 2, -4,  4, -3, -3, -3, -1, -3, -3, -1,  2,  4, -1, -1, -3],
  [ 1,  4, -3, -1, -2, -2,  1, -1,  3, -5, -4, -2, -2, -2, -5],
  [-1, -2,  4,  0,  4,  1,  1,  4, -5,  3,  1, -1,  4,  2, -2],
  [ 4,  3,  2,  4,  4, -5,  2, -5, -5,  2, -5, -4, -4,  0,  3],
  [ 1, -2,  0, -2, -2,  0,  2,  3,  1,  2, -4,  0, -5, -2,  2],
  [-4, -4,  2,  1, -1,  4, -2,  1, -2,  2, -1, -1,  3,  4, -1],
  [-4,  0,  2,  3, -4, -5,  3, -5,  4, -4, -2,  3,  3, -3,  0],
  [ 1,  2,  3, -4,  2,  0, -4, -1,  1, -3,  1,  4, -2,  4, -1],
  [-3,  3,  0,  2,  1, -2, -4,  0, -1, -1, -1,  2,  3,  1, -4],
  [ 4,  3, -3, -4,  3,  1, -3,  2,  1, -5, -5, -3,  2,  1,  4],
  [-3, -5, -1, -5, -2,  0, -3,  1,  2, -1,  0, -4,  3, -2,  3],
  [-1,  1, -3, -1,  3, -3,  2, -3, -5, -1, -1,  3, -1,  2,  3]
                    ])
B.determinant()
```

```
202905135564
```

Sage is incredibly fast at computing determinants with rational entries. Try the following two compute cells on whatever computer you might be using right now. The one unfamilar command clears the value of the determinant that Sage caches, so we get accurate timing information across multiple evaluations.

```
C = random_matrix(QQ, 100, 100)
timeit("C.determinant(); C._cache={}")      # random
```

```
5 loops, best of 3: 152 ms per loop
```

```
C.determinant()                             # random
```

```
-54987836999...175801344
```

Section PDM: Properties of Determinants of Matrices

NME7 Nonsingular Matrices, Round 7

Our latest addition to the NME series is sometimes considered the best computational way to determine if a matrix is nonsingular. While computing a single number for a comparison to zero is a very appealing process, other properties of nonsingular matrices are faster for determining if a matrix is nonsingular. Still, it is a worthwhile equivalence. We illustrate both directions.

```
A = matrix(QQ, [[ 1,  0,  2,  1,  1, -3,  4, -6],
                [ 0,  1,  0,  5,  3, -8,  0,  6],
                [ 1,  1,  4,  4,  2, -8,  4, -2],
                [ 0,  1,  6,  0, -2, -1,  1,  0],
                [-1,  1,  0,  1,  0, -2, -1,  5],
                [ 0, -1, -6, -3,  1,  4,  1, -5],
                [-1, -1, -2, -3, -2,  7, -4,  2],
                [ 0,  2,  0,  0, -2, -2,  0,  6]])
A.is_singular()
```

```
False
```

```
A.determinant()
```

```
4
```

```
B = matrix(QQ, [[ 2, -1,  0,  4, -6, -2,  8, -2],
                [-1,  1, -1, -4, -1,  7, -7, -2],
                [-1,  1,  1,  0,  7, -5, -6,  7],
                [ 2, -1, -1,  3, -5, -2,  5,  2],
                [ 1, -2,  0,  2, -1, -3,  8,  0],
                [-1,  1,  0, -2,  6, -1, -8,  5],
                [ 2, -1, -1,  2, -7,  1,  7, -3],
                [-1,  0,  1,  0,  6, -4, -2,  4]])
B.is_singular()
```

```
True
```

```
B.determinant()
```

```
0
```

PDM Properties of Determinants of Matrices

There are many properties of determinants detailed in this section. You have the tools in Sage to explore all of them. We will illustrate with just two. One is not particularly deep, but we personally find the interplay between matrix multiplication and the determinant nothing short of amazing, so we will not pass up the chance to verify Theorem DRMM. Add some checks of other properties yourself.

We copy the sixth row of the matrix `C` to the fourth row, so by Theorem DERC, the determinant is zero.

```
C = matrix(QQ, [[-3,  4,  0,  1,  2,  0,  5,  0],
                [ 4,  0, -4,  7,  0,  5,  0,  5],
                [ 7,  4,  2,  2,  4,  0, -2,  8],
                [ 5, -4,  8,  2,  6, -1, -4, -4],
                [ 8,  0,  7,  4,  7,  5,  2, -3],
                [ 6,  5,  3,  7,  4,  2,  4, -3],
                [ 1,  6, -4, -4,  3,  8,  5, -2],
                [ 2,  4, -2,  8,  2,  5,  2,  2]])
C[3] = C[5]
C
```

```
[-3  4  0  1  2  0  5  0]
[ 4  0 -4  7  0  5  0  5]
[ 7  4  2  2  4  0 -2  8]
[ 6  5  3  7  4  2  4 -3]
[ 8  0  7  4  7  5  2 -3]
[ 6  5  3  7  4  2  4 -3]
[ 1  6 -4 -4  3  8  5 -2]
[ 2  4 -2  8  2  5  2  2]
```

```
C.determinant()
```

```
0
```

Now, a demonstration of Theorem DRMM.

```
A = matrix(QQ, [[-4,  7, -2,  6,  8,  0, -4,  6],
                [ 1,  6,  5,  8,  2,  3,  1, -1],
                [ 5,  0,  7, -3,  7, -3,  6, -3],
                [-4,  5,  8,  3,  6,  8, -1, -1],
                [ 0,  0, -3, -3,  4,  4,  2,  5],
                [-3, -2, -3,  8,  8, -3, -1,  1],
                [-4,  0,  2,  4,  4,  4,  7,  2],
                [ 3,  3, -4,  5, -2,  3, -1,  5]])
B = matrix(QQ, [[ 1,  2, -4,  8, -1,  2, -1, -3],
                [ 3,  3, -2, -4, -3,  8,  1,  6],
                [ 1,  8,  4,  0,  4, -2,  0,  8],
                [ 6,  8,  1, -1, -4, -3, -2,  5],
                [ 0,  5,  1,  4, -3,  2, -3, -2],
                [ 2,  4,  0,  7,  8, -1,  5,  8],
                [ 7,  1,  1, -1, -1,  7, -2,  6],
                [ 2,  3,  4,  7,  3,  4,  7, -2]])
C = A*B; C
```

```
[ 35  99  28  12 -51  46  25  16]
[ 83 144  11  -3 -17  20 -11 141]
[ 24  62   6  23 -25  52 -68  30]
[ 44 153  42  19  53   0  20 169]
[ 11   5  11  80  33  53  45 -13]
[ 25  58  23  -5 -79 -24 -45 -35]
[ 83  89  47  15  15  37   4 110]
[ 47  39 -12  56  -2  29  48  14]
```

```
Adet = A.determinant(); Adet
```

```
9350730
```

```
Bdet = B.determinant(); Bdet
```

```
6516260
```

```
Cdet = C.determinant(); Cdet
```

```
60931787869800
```

```
Adet*Bdet == Cdet
```

```
True
```

Earlier, in Sage NME1 we used the `random_matrix()` constructor with the `algorithm='unimodular'` keyword to create a nonsingular matrix. We can now reveal that in this context, "unimodular" means "with determinant 1." So such a matrix will always be nonsingular by Theorem SMZD. But more can be said. It is not obvious at all, and Solution PDM.SOL.M20 has just a partial explanation, but the inverse of a unimodular matrix with all integer entries will have an inverse with all integer entries.

With a fraction-free inverse many "textbook" exercises can be constructed through the use of unimodular matrices. Experiment for yourself below. An `upper_bound` keyword can be used to control the size of the entries of the matrix, though this will have little control over the inverse.

```
A = random_matrix(QQ, 5, algorithm='unimodular')
A, A.inverse()                          # random
```

```
(
[  -9  -32 -118  273   78]  [-186   30   22  -18  375]
[   2    9   31  -69  -21]  [  52   -8   -8    3 -105]
[   4   15   54 -120  -39]  [-147   25   19  -12  297]
[  -3  -11  -38   84   28]  [ -47    8    6   -4   95]
[  -5  -18  -66  152   44], [ -58   10    7   -5  117]
)
```

Chapter E
Eigenvalues

Section EE: Eigenvalues and Eigenvectors

EE Eigenvalues and Eigenvectors

Sage can compute eigenvalues and eigenvectors of matrices. We will see shortly that there are subtleties involved with using these routines, but here is a quick example to begin with. These two commands should be enough to get you started with most of the early examples in this section. See the end of the section for more comprehensive advice.

For a square matrix, the methods `.eigenvalues()` and `.eigenvectors_right()` will produce what you expect, though the format of the eigenvector output requires some explanation. Here is Example SEE from the start of this chapter.

```
A = matrix(QQ, [[ 204,   98, -26, -10],
                [-280, -134,  36,  14],
                [ 716,  348, -90, -36],
                [-472, -232,  60,  28]])
A.eigenvalues()
```

```
[4, 0, 2, 2]
```

```
A.eigenvectors_right()
```

```
[(4, [
(1, -1, 2, 5)
], 1), (0, [
(1, -4/3, 10/3, -4/3)
], 1), (2, [
(1, 0, 7, 2),
(0, 1, 3, 2)
], 2)]
```

The three eigenvalues we know are included in the output of `eigenvalues()`, though for some reason the eigenvalue $\lambda = 2$ shows up twice.

The output of the `eigenvectors_right()` method is a list of triples. Each triple begins with an eigenvalue. This is followed by a list of eigenvectors for that eigenvalue. Notice the first eigenvector is identical to the one we described in Example SEE. The eigenvector for $\lambda = 0$ is different, but is just a scalar multiple of the one from Example SEE. For $\lambda = 2$, we now get two eigenvectors, and neither looks like either of the ones from Example SEE. (Hint: try writing the eigenvectors from the example as linear combinations of the two in the Sage output.) An explanation of

the the third part of each triple (an integer) will have to wait, though it can be optionally surpressed if desired.

One cautionary note: The word `lambda` has a special purpose in Sage, so do not try to use this as a name for your eigenvalues.

CEVAL Computing Eigenvalues

We can now give a more careful explantion about eigenvalues in Sage. Sage will compute the characteristic polynomial of a matrix, with amazing ease (in other words, quite quickly, even for large matrices). The two matrix methods `.charpoly()` and `.characteristic_polynomial()` do exactly the same thing. We will use the longer name just to be more readable, you may prefer the shorter.

We now can appreciate a very fundamental obstacle to determining the eigenvalues of a matrix, which is a theme that will run through any advanced study of linear algebra. Study this example carefully before reading the discussion that follows.

```
A = matrix(QQ, [[-24,   6,  0,  -1,  31,   7],
                [ -9,  -2, -8, -17,  24, -29],
                [  4, -10,  1,   1, -12, -36],
                [-19,  11, -1,  -4,  33,  29],
                [-11,   6,  2,   3,  14,  21],
                [  5,  -1,  2,   5, -11,   4]])
A.characteristic_polynomial()
```

```
x^6 + 11*x^5 + 15*x^4 - 84*x^3 - 157*x^2 + 124*x + 246
```

```
A.characteristic_polynomial().factor()
```

```
(x + 3) * (x^2 - 2) * (x^3 + 8*x^2 - 7*x - 41)
```

```
B = A.change_ring(QQbar)
B.characteristic_polynomial()
```

```
x^6 + 11*x^5 + 15*x^4 - 84*x^3 - 157*x^2 + 124*x + 246
```

```
B.characteristic_polynomial().factor()
```

```
(x - 2.356181157637288?) * (x - 1.414213562373095?) *
(x + 1.414213562373095?) * (x + 2.110260216209409?) *
(x + 3) * (x + 8.245920941427888?)
```

We know by Theorem EMRCP that to compute eigenvalues, we need the roots of the characteristic polynomial, and from basic algebra, we know these correspond to linear factors. However, with our matrix defined with entries from `QQ`, the factorization of the characteristic polynomial does not "leave" that number system, only factoring "far enough" to retain factors with rational coefficients. The solutions to $x^2 - 2 = 0$ are somewhat obvious ($\pm\sqrt{2} \approx \pm 1.414213$), but the roots of the cubic factor are more obscure.

But then we have `QQbar` to the rescue. Since this number system contains the roots of every possible polynomial with integer coefficients, we can totally factor

any characteristic polynomial that results from a matrix with entries from `QQbar`. A common situation will be to begin with a matrix having rational entries, yet the matrix has a characteristic polynomial with roots that are complex numbers.

We can demonstrate this behavior with the `extend` keyword option, which tells Sage whether or not to expand the number system to contain the eigenvalues.

```
A.eigenvalues(extend=False)
```

```
[-3]
```

```
A.eigenvalues(extend=True)
```

```
[-3, -1.4142135623730950?, 1.4142135623730950?,
 -8.245920941427880?, -2.110260216209409?, 2.356181157637288?]
```

For matrices with entries from `QQ`, the default behavior is to extend to `QQbar` when necessary. But for other number systems, you may need to explicitly use the `extend=True` option.

From a factorization of the characteristic polynomial, we can see the algebraic multiplicity of each eigenvalue as the second entry of the each pair returned in the list. We demonstrate with Example SEE, extending to `QQbar`, which is not strictly necessary for this simple matrix.

```
A = matrix(QQ, [[204, 98, -26, -10],
                [-280, -134, 36, 14],
                [716, 348, -90, -36],
                [-472, -232, 60, 28]])
A.characteristic_polynomial().roots(QQbar)
```

```
[(0, 1), (2, 2), (4, 1)]
```

One more example, which illustrates the behavior when we use floating-point approximations as entries (in other words, we use `CDF` as our number system). This is Example EMMS4, both as an exact matrix with entries from `QQbar` and as an approximate matrix with entries from `CDF`.

```
A = matrix(QQ, [[-2,  1, -2, -4],
                [12,  1,  4,  9],
                [ 6,  5, -2, -4],
                [ 3, -4,  5, 10]])
A.eigenvalues()
```

```
[1, 2, 2, 2]
```

```
B = A.change_ring(CDF)
B.eigenvalues()
```

```
[2.00001113832 + 1.75245331504e-05*I,
2.00000960797 - 1.84081096655e-05*I,
1.99997925371 + 8.8357651007e-07*I,
1.0]
```

So, we see $\lambda = 2$ as an eigenvalue with algebraic multiplicity 3, while the numerical results contain three complex numbers, each very, very close to 2. The approximate nature of these eigenvalues may be disturbing (or alarming). However, their computation, as floating-point numbers, can be incredibly fast with sophisticated algorithms allowing the analysis of huge matrices with millions of entries. And perhaps your original matrix includes data from an experiment, and is not even exact in the first place. Designing and analyzing algorithms to perform these computations quickly and accurately is part of the field known as numerical linear algebra.

One cautionary note: Sage uses a definition of the characteristic polynomial slightly different than ours, namely $\det(xI_n - A)$. This has the advantage that the x^n term always has a positive one as the leading coefficient. For even values of n the two definitions create the identical polynomial, and for odd values of n, the two polynomials differ only by a multiple of -1. The reason this is not very critical is that Theorem EMRCP is true in either case, and this is a principal use of the characteristic polynomial. Our definition is more amenable to computations by hand.

CEVEC Computing Eigenvectors

There are three ways to get eigenvectors in Sage. For each eigenvalue, the method `.eigenvectors_right()` will return a list of eigenvectors that is a basis for the associated eigenspace. The method `.eigenspaces_right()` will return an eigenspace (in other words, a vector space, rather than a list of vectors) for each eigenvalue. There are also `eigenmatrix` methods which we will describe at the end of the chapter in Sage MD.

The matrix method `.eigenvectors_right()` (or equivalently the matrix method `.right_eigenvectors()`) produces a list of triples, one triple per eigenvalue. Each triple has an eigenvalue, a list, and then the algebraic multiplicity of the eigenvalue. The list contains vectors forming a basis for the eigenspace. Notice that the length of the list of eigenvectors will be the geometric multiplicity (and there is no easier way to get this information).

```
A = matrix(QQ, [[-5,  1, 7, -15],
                [-2,  0, 5,  -7],
                [ 1, -5, 7,  -3],
                [ 4, -7, 3,   4]])
A.eigenvectors_right()
```

```
[(3, [
(1, 0, -1, -1),
(0, 1, 2, 1)
], 2),
(-2*I, [(1, 0.3513513513513514? + 0.1081081081081081?*I,
0.02702702702702703? + 0.1621621621621622?*I,
-0.2972972972972973? + 0.2162162162162163?*I)], 1),
(2*I, [(1, 0.3513513513513514? - 0.1081081081081081?*I,
0.02702702702702703? - 0.1621621621621622?*I,
-0.2972972972972973? - 0.2162162162162163?*I)], 1)]
```

Note that this is a good place to practice burrowing down into Sage output that is full of lists (and lists of lists). See if you can extract just the second eigenvector

for $\lambda = 3$ using a single statement. Or perhaps try obtaining the geometric multiplicity of $\lambda = -2i$. Notice, too, that Sage has automatically upgraded to `QQbar` to get the complex eigenvalues.

The matrix method `.eigenspaces_right()` (equal to `.right_eigenspaces()`) produces a list of pairs, one pair per eigenvalue. Each pair has an eigenvalue, followed by the eigenvalue's eigenspace. Notice that the basis matrix of the eigenspace may not have the same eigenvectors you might get from other methods. Similar to the `eigenvectors` method, the dimension of the eigenspace will yield the geometric multiplicity (and there is no easier way to get this information). If you need the algebraic multiplicities, you can supply the keyword option `algebraic_multiplicity=True` to get back triples with the algebraic multiplicity in the third entry of the triple. We will recycle the example above, and not demonstrate the algebraic multiplicity option. (We have formatted the one-row basis matrices over `QQbar` across several lines.)

```
A.eigenspaces_right()
```

```
[
(3, Vector space of degree 4 and dimension 2 over Rational Field
User basis matrix:
[ 1  0 -1 -1]
[ 0  1  2  1]),
(-2*I, Vector space of degree 4 and dimension 1 over Algebraic Field
User basis matrix:
[                                          1
 0.3513513513513514?  + 0.1081081081081081?*I
 0.02702702702702703? +  0.1621621621621622?*I
 -0.2972972972972973? +  0.2162162162162163?*I]),
(2*I, Vector space of degree 4 and dimension 1 over Algebraic Field
User basis matrix:
[                                          1
 0.3513513513513514?  - 0.1081081081081081?*I
 0.02702702702702703? -  0.1621621621621622?*I
-0.2972972972972973?  -  0.2162162162162163?*I])
]
```

Notice how the output includes a subspace of dimension two over the rationals, and two subspaces of dimension one over the algebraic numbers.

The upcoming Subsection EE.ECEE has many examples, which mostly reflect techniques that might be possible to verify by hand. Here is the same matrix as above, analyzed in a similar way. Practicing the examples in this subsection, either directly with the higher-level Sage commands, and/or with more primitive commands (as below) would be an extremely good exercise at this point.

```
A = matrix(QQ, [[-5,  1, 7, -15],
                [-2,  0, 5, -7],
                [ 1, -5, 7, -3],
                [ 4, -7, 3, 4]])
# eigenvalues
A.characteristic_polynomial()
```

```
x^4 - 6*x^3 + 13*x^2 - 24*x + 36
```

```
A.characteristic_polynomial().factor()
```

```
(x - 3)^2 * (x^2 + 4)
```

```
A.characteristic_polynomial().roots(QQbar)
```

```
[(3, 2), (-2*I, 1), (2*I, 1)]
```

```
# eigenspaces, rational
(A-3*identity_matrix(4)).right_kernel(basis='pivot')
```

```
Vector space of degree 4 and dimension 2 over Rational Field
User basis matrix:
[ 1  1  1  0]
[-2 -1  0  1]
```

```
# eigenspaces, complex
B = A.change_ring(QQbar)
(B-(2*I)*identity_matrix(4)).right_kernel(basis='pivot')
```

```
Vector space of degree 4 and dimension 1 over Algebraic Field
User basis matrix:
[8/5*I - 11/5  4/5*I - 3/5  2/5*I + 1/5            1]
```

```
(B-(-2*I)*identity_matrix(4)).right_kernel(basis='pivot')
```

```
Vector space of degree 4 and dimension 1 over Algebraic Field
User basis matrix:
[-8/5*I - 11/5  -4/5*I - 3/5  -2/5*I + 1/5             1]
```

Notice how we changed the number system to the algebraic numbers before working with the complex eigenvalues. Also, we are using the `basis='pivot'` keyword option so that bases for the eigenspaces look more like the bases described in Theorem BNS.

By now, it should be clear why we keep using the "right" variants of these methods. Eigenvectors can be defined "on the right", $A\mathbf{x} = \lambda\mathbf{x}$ as we have done, or "on the left," $\mathbf{x}A = \lambda\mathbf{x}$. So use the "right" versions of the eigenvalue and eigenvector commands to stay consistent with the text. Recognize, too, that eigenspaces may be computed with different bases than those given in the text (typically like those for null spaces with the `basis='echelon'` option).

Why does the `.eigenvalues()` method not come in left and right versions? The upcoming Theorem ETM can be used to show that the two versions would have identical output, so there is no need.

Section PEE: Properties of Eigenvalues and Eigenvectors

NME8 Nonsingular Matrices, Round 8

Zero eigenvalues are another marker of singular matrices. We illustrate with two matrices, the first nonsingular, the second not.

```
A = matrix(QQ, [[ 1,  0,  2,  1,  1, -3,  4, -6],
                [ 0,  1,  0,  5,  3, -8,  0,  6],
                [ 1,  1,  4,  4,  2, -8,  4, -2],
                [ 0,  1,  6,  0, -2, -1,  1,  0],
                [-1,  1,  0,  1,  0, -2, -1,  5],
                [ 0, -1, -6, -3,  1,  4,  1, -5],
                [-1, -1, -2, -3, -2,  7, -4,  2],
                [ 0,  2,  0,  0, -2, -2,  0,  6]])
A.is_singular()
```

```
False
```

```
0 in A.eigenvalues()
```

```
False
```

```
B = matrix(QQ, [[ 2, -1,  0,  4, -6, -2,  8, -2],
                [-1,  1, -1, -4, -1,  7, -7, -2],
                [-1,  1,  1,  0,  7, -5, -6,  7],
                [ 2, -1, -1,  3, -5, -2,  5,  2],
                [ 1, -2,  0,  2, -1, -3,  8,  0],
                [-1,  1,  0, -2,  6, -1, -8,  5],
                [ 2, -1, -1,  2, -7,  1,  7, -3],
                [-1,  0,  1,  0,  6, -4, -2,  4]])
B.is_singular()
```

```
True
```

```
0 in B.eigenvalues()
```

```
True
```

Section SD: Similarity and Diagonalization

SM Similar Matrices

It is quite easy to determine if two matrices are similar, using the matrix method `.is_similar()`. However, computationally this can be a very difficult proposition, so support in Sage is incomplete now, though it will always return a result for matrices with rational entries. Here are examples where the two matrices are and are not similar. Notice that the keyword option `transformation=True` will cause a pair to be returned, such that if the matrices are indeed similar, the matrix effecting the similarity transformation will be in the second slot of the pair.

```
A = matrix(QQ, [[ 25,   2,  -8,  -1,  11,  26,  35],
                [ 28,   2, -15,   2,   6,  34,  31],
                [  1, -17, -25,  28, -44,  26, -23],
                [ 36,  -2, -24,  10,  -1,  50,  39],
                [  0,  -7, -13,  14, -21,  14, -11],
                [-22, -17, -16,  27, -51,   1, -53],
                [  -1, 10,  17, -18,  28, -18,  15]])
B = matrix(QQ, [[-89, -16, -55, -23, -104, -48, -67],
                [-15,   1, -20, -21,  -20, -60, -26],
                [ 48,   6,  37,  25,   59,  64,  46],
```

```
                [-96, -16, -49, -16, -114, -23, -67],
                [ 56,  10,  33,  13,   67,  29,  37],
                [ 10,   2,   2,  -2,   12,  -9,   4],
                [ 28,   6,  13,   1,   32,  -4,  16]])
is_sim, trans = A.is_similar(B, transformation=True)
is_sim
```

```
True
```

Since we knew in advance these two matrices are similar, we requested the transformation matrix, so the output is a pair. The similarity matrix is a bit of a mess, so we will use three Sage routines to clean up `trans`. We convert the entries to numerical approximations, clip very small values (less than 10^{-5}) to zero and then round to three decimal places. You can experiment printing just `trans` all by itself.

```
trans.change_ring(RDF).zero_at(10^-5).round(3)
```

```
[   1.0    0.0    0.0    0.0    0.0    0.0    0.0]
[   0.0  1.188  0.375 -0.375  0.562 -0.375  0.375]
[-3.107 -1.072  0.764  1.043 -2.288 -1.758 -2.495]
[ 7.596    0.3 -1.026  -7.29 11.306 -2.705   -1.6]
[ 0.266  0.079 -0.331 -0.387  0.756  0.447  0.429]
[-2.157  0.071  0.469  1.893 -3.014  0.355  0.196]
[-0.625  0.291 -0.068  0.995 -1.315  1.125  1.179]
```

The matrix `C` is not similar to `A` (and hence not similar to `B` by Theorem SER), so we illustrate the return value when we do not request the similarity matrix (since it does not even exist).

```
C = matrix(QQ, [[ 16, -26,  19,  -56,  26,   8, -49],
                [ 20, -43,  36, -102,  52,  23, -65],
                [-18,  29, -21,   62, -30,  -9,  56],
                [-17,  31, -27,   73, -37, -16,  49],
                [ 18, -36,  30,  -82,  43,  18, -54],
                [-32,  62, -56,  146, -77, -35,  88],
                [ 11, -19,  17,  -44,  23,  10, -29]])
C.is_similar(A)
```

```
False
```

MD Matrix Diagonalization

The third way to get eigenvectors is the matrix method `.eigenmatrix_right()` (and the analogous `.eigenmatrix_left()`). It always returns two square matrices of the same size as the original matrix. The first matrix of the output is a diagonal matrix with the eigenvalues of the matrix filling the diagonal entries of the matrix. The second matrix has eigenvectors in the columns, in the same order as the corresponding eigenvalues. For a single eigenvalue, these columns/eigenvectors form a linearly independent set.

A careful reading of the previous paragraph suggests the question: what if we do not have enough eigenvectors to fill the columns of the second square matrix? When the geometric multiplicity does not equal the algebraic multiplicity, the deficit is met by inserting zero columns in the matrix of eigenvectors. Conversely, when the matrix is diagonalizable, by Theorem DMFE the geometric and algebraic multiplicities of each eigenvalue are equal, and the union of the bases of the eigenspaces provides a

complete set of linearly independent vectors. So for a matrix A, Sage will output two matrices, D and S such that $S^{-1}AS = D$.

We can rewrite the relation above as $AS = SD$. In the case of a non-diagonalizable matrix, the matrix of eigenvectors is singular (it has zero columns), but the relationship $AS = SD$ still holds. Here are examples of the two scenarios, along with demonstrations of the matrix method `is_diagonalizable()`.

```
A = matrix(QQ, [[ 2, -18, -68,  64, -99, -87,  83],
                [ 4, -10, -41,  34, -58, -57,  46],
                [ 4,  16,  59, -60,  86,  70, -72],
                [ 2, -15, -65,  57, -92, -81,  78],
                [-4,  -7, -32,  31, -45, -31,  41],
                [ 2,  -6, -22,  20, -32, -31,  26],
                [ 0,   7,  30, -27,  42,  37, -36]])
D, S = A.eigenmatrix_right()
D
```

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

```
S
```

```
[     1      1      0      1      0      1      0]
[ 13/23      0      1      0      1      0      1]
[-20/23    1/3     -2     -1    2/7   -4/3    5/6]
[ 21/23    1/3      1      0   10/7      1      0]
[ 10/23   -1/6      1     -1   15/7    4/3   -4/3]
[  8/23    1/3      0      1     -1      0    1/2]
[-10/23    1/6     -1     -1    6/7   -1/3   -1/6]
```

```
S.inverse()*A*S == D
```

```
True
```

```
A.is_diagonalizable()
```

```
True
```

Now for a matrix that is far from diagonalizable.

```
B = matrix(QQ, [[ 37, -13,  30,  81, -74, -13,  18],
                [  6,  26,  21, -11, -46, -48,  19],
                [ 16,  10,  29,  16, -42, -39,  26],
                [-24,   8, -24, -53,  54,  13, -15],
                [ -8,   3,  -8, -20,  24,   4,  -5],
                [ 31,  12,  46,  48, -97, -56,  35],
                [  8,   5,  16,  12, -34, -20,  11]])
```

```
D, S = B.eigenmatrix_right()
D
```

```
[-2  0  0  0  0  0  0]
[ 0 -2  0  0  0  0  0]
[ 0  0 -2  0  0  0  0]
[ 0  0  0  6  0  0  0]
[ 0  0  0  0  6  0  0]
[ 0  0  0  0  0  6  0]
[ 0  0  0  0  0  0  6]
```

```
S
```

```
[   1    0    0    1    0    0    0]
[ 1/4    0    0  1/4    0    0    0]
[ 1/2    0    0  3/8    0    0    0]
[-3/4    0    0 -5/8    0    0    0]
[-1/4    0    0 -1/4    0    0    0]
[   1    0    0  7/8    0    0    0]
[ 1/4    0    0  1/4    0    0    0]
```

```
B*S == S*D
```

```
True
```

```
B.is_diagonalizable()
```

```
False
```

Chapter LT
Linear Transformations

Section LT: Linear Transformations

LTS Linear Transformations, Symbolic

There are several ways to create a linear transformation in Sage, and many natural operations on these objects are possible. As previously, rather than use the complex numbers as our number system, we will instead use the rational numbers, since Sage can model them exactly. We will also use the following transformation repeatedly for examples, when no special properties are required:

$$T\colon \mathbb{C}^3 \to \mathbb{C}^2$$
$$T\left(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}\right) = \begin{bmatrix} -x_1 + 2x_3 \\ x_1 + 3x_2 + 7x_3 \\ x_1 + x_2 + x_3 \\ 2x_1 + 3x_2 + 5x_3 \end{bmatrix}$$

To create this linear transformation in Sage, we first create a "symbolic" function, which requires that we also first define some symbolic variables which are `x1`, `x2` and `x3` in this case. (You can bypass the intermediate variable `outputs` in your own work. We will use it consistently to allow us to spread the definition across several lines without the Sage preparser getting in the way. In other words, it is safe to combine the the two lines below and not use `outputs`.)

```
x1, x2, x3 = var('x1, x2, x3')
outputs = [ -x1        + 2*x3,
             x1 + 3*x2 + 7*x3,
             x1 +   x2 +   x3,
           2*x1 + 3*x2 + 5*x3]
T_symbolic(x1, x2, x3) = outputs
```

You can experiment with `T_symbolic`, evaluating it at triples of rational numbers, and perhaps doing things like calculating its partial derivatives. We will use it as input to the `linear_transformation()` constructor. We just need to specify carefully the domain and codomain, now as vector spaces over the rationals rather than the complex numbers.

```
T = linear_transformation(QQ^3, QQ^4, T_symbolic)
```

You can now, of course, experiment with `T` via tab-completion, but we will explain the various properties of Sage linear transformations as we work through this chapter. Even some seemingly simple operations, such as printing `T` will require some explanation. But for starters, we can evaluate `T`.

```
u = vector(QQ, [3, -1, 2])
w = T(u)
w
```

```
(1, 14, 4, 13)
```

```
w.parent()
```

```
Vector space of dimension 4 over Rational Field
```

Here is a small verification of Theorem LTTZZ.

```
zero_in = zero_vector(QQ, 3)
zero_out = zero_vector(QQ, 4)
T(zero_in) == zero_out
```

```
True
```

Note that Sage will recognize some symbolic functions as not being linear transformations (in other words, inconsistent with Definition LT), but this detection is fairly easy to fool. We will see some safer ways to create a linear transformation shortly.

LTM Linear Transformations, Matrices

A second way to build a linear transformation is to use a matrix, as motivated by Theorem MBLT. But there is one caveat. We have seen that Sage has a preference for rows, so when defining a linear transformation with a product of a matrix and a vector, *Sage forms a linear combination of the rows of the matrix with the scalars of the vector*. This is expressed by writing the vector on the left of the matrix, where if we were to interpret the vector as a 1-row matrix, then the definition of matrix multiplication would do the right thing.

Remember throughout, a linear transformation has very little to do with the mechanism by which we define it. Whether or not we use matrix multiplication with vectors on the left (Sage internally) or matrix multiplication with vectors on the right (your text), what matters is the *function* that results. One concession to the "vector on the right" approach is that we can tell Sage that we mean for the matrix to define the linear transformation by multiplication with the vector on the right. Here is our running example again — with some explanation following.

```
A = matrix(QQ, [[-1, 0, 2],
                [ 1, 3, 7],
                [ 1, 1, 1],
                [ 2, 3, 5]])
T = linear_transformation(QQ^3, QQ^4, A, side='right')
T
```

```
Vector space morphism represented by the matrix:
[-1  1  1  2]
[ 0  3  1  3]
[ 2  7  1  5]
Domain: Vector space of dimension 3 over Rational Field
Codomain: Vector space of dimension 4 over Rational Field
```

The way `T` prints reflects the way Sage carries `T` internally. But notice that we defined `T` in a way that is consistent with the text, via the use of the optional `side='right'` keyword. If you rework examples from the text, or use Sage to assist with exercises, be sure to remember this option. In particular, when the matrix is square it might be easy to miss that you have forgotten this option. Note too that Sage uses a more general term for a linear transformation, "vector space morphism." Just mentally translate from Sage-speak to the terms we use here in the text.

If the standard way that Sage prints a linear transformation is too confusing, you can get all the relevant information with a handful of commands.

```
T.matrix(side='right')
```

```
[-1  0  2]
[ 1  3  7]
[ 1  1  1]
[ 2  3  5]
```

```
T.domain()
```

```
Vector space of dimension 3 over Rational Field
```

```
T.codomain()
```

```
Vector space of dimension 4 over Rational Field
```

So we can build a linear transformation in Sage from a matrix, as promised by Theorem MBLT. Furthermore, Theorem MLTCV says there is a matrix associated with every linear transformation. This matrix is provided in Sage by the `.matrix()` method, where we use the option `side='right'` to be consistent with the text. Here is Example MOLT reprised, where we define the linear transformation via a Sage symbolic function and then recover the matrix of the linear transformation.

```
x1, x2, x3 = var('x1, x2, x3')
outputs = [3*x1 - 2*x2 + 5*x3,
             x1 +   x2 +   x3,
           9*x1 - 2*x2 + 5*x3,
                  4*x2       ]
S_symbolic(x1, x2, x3) = outputs
S = linear_transformation(QQ^3, QQ^4, S_symbolic)
C = S.matrix(side='right'); C
```

```
[ 3 -2  5]
[ 1  1  1]
[ 9 -2  5]
[ 0  4  0]
```

```
x = vector(QQ, [2, -3, 3])
S(x) == C*x
```

```
True
```

LTB Linear Transformations, Bases

A third way to create a linear transformation in Sage is to provide a list of images for a basis, as motivated by Theorem LTDB. The default is to use the standard basis as the inputs (Definition SUV). We will, once again, create our running example.

```
U = QQ^3
V = QQ^4
v1 = vector(QQ, [-1, 1, 1, 2])
v2 = vector(QQ, [ 0, 3, 1, 3])
v3 = vector(QQ, [ 2, 7, 1, 5])
T = linear_transformation(U, V, [v1, v2, v3])
T
```

```
Vector space morphism represented by the matrix:
[-1  1  1  2]
[ 0  3  1  3]
[ 2  7  1  5]
Domain: Vector space of dimension 3 over Rational Field
Codomain: Vector space of dimension 4 over Rational Field
```

Notice that there is no requirement that the list of images (in Sage or in Theorem LTDB) is a basis. They do not even have to be different. They could all be the zero vector (try it).

If we want to use an alternate basis for the domain, it is possible, but there are two caveats. The first caveat is that we must be sure to provide a basis for the domain, Sage will give an error if the proposed basis is not linearly independent and we are responsible for providing the right number of vectors (which should be easy).

We have seen that vector spaces can have alternate bases, which prints as a "user basis." Here will provide the domain with an alternate basis. The relevant command will create a subspace, but for now, we need to provide a big enough set to create the entire domain. It is possible to use fewer linearly independent vectors, and create a proper subspace, but then we will not be able to use this proper subspace to build the linear transformation we want.

```
u1 = vector(QQ, [ 1,  3, -4])
u2 = vector(QQ, [-1, -2,  3])
u3 = vector(QQ, [ 1,  1, -3])
U = (QQ^3).subspace_with_basis([u1, u2, u3])
U == QQ^3
```

```
True
```

```
U.basis_matrix()
```

```
[ 1  3 -4]
[-1 -2  3]
[ 1  1 -3]
```

```
U.echelonized_basis_matrix()
```

```
[1 0 0]
[0 1 0]
[0 0 1]
```

We can use this alternate version of `U` to create a linear transformation from specified images. Superficially there is nothing real special about our choices for `v1, v2, v3`.

```
V = QQ^4
v1 = vector(QQ, [-9, -18,  0, -9])
v2 = vector(QQ, [ 7,  14,  0,  7])
v3 = vector(QQ, [-7, -17, -1, -10])
```

Now we create the linear transformation. Here is the second caveat: the matrix of the linear transformation is no longer that provided by Theorem MLTCV. It may be obvious where the matrix comes from, but a full understanding of its interpretation will have to wait until Section MR.

```
S = linear_transformation(U, V, [v1, v2, v3])
S.matrix(side='right')
```

```
[ -9   7  -7]
[-18  14 -17]
[  0   0  -1]
[ -9   7 -10]
```

We suggested our choices for `v1, v2, v3` were "random." Not so — the linear transformation `S` just created is equal to the linear transformation `T` above. If you have run all the input in this subsection, in order, then you should be able to compare the *functions* `S` and `T`. The next command should *always* produce `True`.

```
u = random_vector(QQ, 3)
T(u) == S(u)
```

```
True
```

Notice that `T == S` may not do what you expect here. Instead, the linear transformation method `.is_equal_function()` will perform a conclusive check of equality of two linear transformations as functions.

```
T.is_equal_function(S)
```

```
True
```

Can you reproduce this example? In other words, define some linear transformation, any way you like. Then give the domain an alternate basis and concoct the correct images to create a second linear transformation (by the method of this subsection) which is equal to the first.

PI Pre-Images

Sage handles pre-images just a bit differently than our approach in the text. For the moment, we can obtain a single vector in the set that is the pre-image via the `.preimage_representative()` method. Understand that this method will return *just one* element of the pre-image set, and we have no real control over which one. Also, it is certainly possible that a pre-image is the empty set — in this case, the method will raise a `ValueError`. We will use our running example to illustrate.

```
A = matrix(QQ, [[-1, 0, 2],
                [ 1, 3, 7],
                [ 1, 1, 1],
                [ 2, 3, 5]])
T = linear_transformation(QQ^3, QQ^4, A, side='right')
v = vector(QQ, [1, 2, 0, 1])
u = T.preimage_representative(v)
u
```

```
(-1, 1, 0)
```

```
T(u) == v
```

```
True
```

```
T.preimage_representative(vector(QQ, [1, 2, 1, 1]))
```

```
Traceback (most recent call last):
...
ValueError: element is not in the image
```

Remember, we have defined the pre-image as a set, and Sage just gives us a single element of the set. We will see in Sage ILT that the upcoming Theorem KPI explains why this is no great shortcoming in Sage.

OLT Operations on Linear Transformations

It is possible in Sage to add linear transformations (Definition LTA), multiply them by scalars (Definition LTSM) and compose (Definition LTC) them. Then Theorem SLTLT Theorem MLTLT, and Theorem CLTLT (respectively) tell us the results are again linear transformations. Here are some examples:

```
U = QQ^4
V = QQ^2
A = matrix(QQ, 2, 4, [[-1, 3, 4,  5],
                      [ 2, 0, 3, -1]])
T = linear_transformation(U, V, A, side='right')
B = matrix(QQ, 2, 4, [[-7, 4, -2,  0],
                      [ 1, 1,  8, -3]])
S = linear_transformation(U, V, B, side='right')
P = S + T
P
```

```
Vector space morphism represented by the matrix:
[-8  3]
[ 7  1]
[ 2 11]
[ 5 -4]
Domain: Vector space of dimension 4 over Rational Field
Codomain: Vector space of dimension 2 over Rational Field
```

```
Q = S*5
Q
```

```
Vector space morphism represented by the matrix:
[-35   5]
[ 20   5]
[-10  40]
[  0 -15]
Domain: Vector space of dimension 4 over Rational Field
Codomain: Vector space of dimension 2 over Rational Field
```

Perhaps the only surprise in all this is the necessity of writing scalar multiplication on the right of the linear transformation (rather on the left, as we do in the text). We will recycle the linear transformation `T` from above and redefine `S` to form an example of composition.

```
W = QQ^3
C = matrix(QQ, [[ 4, -2],
                [-1,  3],
                [-3,  2]])
S = linear_transformation(V, W, C, side='right')
R = S*T
R
```

```
Vector space morphism represented by the matrix:
[ -8   7   7]
[ 12  -3  -9]
[ 10   5  -6]
[ 22  -8 -17]
Domain: Vector space of dimension 4 over Rational Field
Codomain: Vector space of dimension 3 over Rational Field
```

We use the star symbol (`*`) to indicate composition of linear transformations. Notice that the order of the two linear transformations we compose is important, and Sage's order agrees with the text. The order does not have to agree, and there are good arguments to have it reversed, so be careful if you read about composition elsewhere.

This is a good place to expand on Theorem VSLT, which says that with definitions of addition and scalar multiplication of linear transformations we then arrive at a vector space. A vector space full of linear transformations. Objects in Sage have "parents" — vectors have vector spaces for parents, fractions of integers have the rationals as parents. What is the parent of a linear transformation? Let us see, by investigating the parent of `S` just defined above.

```
P = S.parent()
P
```

```
Set of Morphisms (Linear Transformations) from
Vector space of dimension 2 over Rational Field to
Vector space of dimension 3 over Rational Field
```

"Morphism" is a general term for a function that "preserves structure" or "respects operations." In Sage a collection of morphisms is referenced as a "homset" or a "homspace." In this example, we have a homset that is the vector space of linear transformations that go from a dimension 2 vector space over the rationals to a dimension 3 vector space over the rationals. What can we do with it? A few things, but not everything you might imagine. It does have a basis, containing a few very simple linear transformations:

```
P.basis()
```

```
(Vector space morphism represented by the matrix:
[1 0 0]
[0 0 0]
Domain: Vector space of dimension 2 over Rational Field
Codomain: Vector space of dimension 3 over Rational Field,
Vector space morphism represented by the matrix:
[0 1 0]
[0 0 0]
Domain: Vector space of dimension 2 over Rational Field
Codomain: Vector space of dimension 3 over Rational Field,
Vector space morphism represented by the matrix:
[0 0 1]
[0 0 0]
Domain: Vector space of dimension 2 over Rational Field
Codomain: Vector space of dimension 3 over Rational Field,
Vector space morphism represented by the matrix:
[0 0 0]
[1 0 0]
Domain: Vector space of dimension 2 over Rational Field
Codomain: Vector space of dimension 3 over Rational Field,
Vector space morphism represented by the matrix:
[0 0 0]
[0 1 0]
Domain: Vector space of dimension 2 over Rational Field
Codomain: Vector space of dimension 3 over Rational Field,
Vector space morphism represented by the matrix:
[0 0 0]
[0 0 1]
Domain: Vector space of dimension 2 over Rational Field
Codomain: Vector space of dimension 3 over Rational Field)
```

You can create a set of linear transformations with the `Hom()` function, simply by giving the domain and codomain.

```
H = Hom(QQ^6, QQ^9)
H
```

```
Set of Morphisms (Linear Transformations) from
Vector space of dimension 6 over Rational Field to
Vector space of dimension 9 over Rational Field
```

An understanding of Sage's homsets is not critical to understanding the use of Sage during the remainder of this course. But such an understanding can be very useful in understanding some of Sage's more advanced and powerful features.

Section ILT: Injective Linear Transformations

ILT Injective Linear Transformations

By now, you have probably already figured out how to determine if a linear transformation is injective, and what its kernel is. You may also now begin to understand why Sage calls the null space of a matrix a kernel. Here are two examples, first a reprise of Example NKAO.

```
U = QQ^3
V = QQ^5
x1, x2, x3 = var('x1, x2, x3')
outputs = [ -x1 +   x2 - 3*x3,
            -x1 + 2*x2 - 4*x3,
             x1 +   x2 +   x3,
           2*x1 + 3*x2 +   x3,
             x1        + 2*x3]
T_symbolic(x1, x2, x3) = outputs
T = linear_transformation(U, V, T_symbolic)
T.is_injective()
```

```
False
```

```
T.kernel()
```

```
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[   1 -1/2 -1/2]
```

So we have a concrete demonstration of one half of Theorem KILT. Here is the second example, a do-over for Example TKAP, but renamed as S.

```
U = QQ^3
V = QQ^5
x1, x2, x3 = var('x1, x2, x3')
outputs = [  -x1 +   x2 +   x3,
             -x1 + 2*x2 + 2*x3,
              x1 +   x2 + 3*x3,
            2*x1 + 3*x2 +   x3,
           -2*x1 +   x2 + 3*x3]
S_symbolic(x1, x2, x3) = outputs
S = linear_transformation(U, V, S_symbolic)
S.is_injective()
```

```
True
```

```
S.kernel()
```

```
Vector space of degree 3 and dimension 0 over Rational Field
Basis matrix:
[]
```

```
S.kernel() == U.subspace([])
```

```
True
```

And so we have a concrete demonstration of the other half of Theorem KILT.

Now that we have Theorem KPI, we can return to our discussion from Sage PI. The `.preimage_representative()` method of a linear transformation will give us a *single* element of the pre-image, with no other guarantee about the nature of that element. That is fine, since this is all Theorem KPI requires (in addition to the kernel). Remember that not every element of the codomain may have a nonempty pre-image (as indicated in the hypotheses of Theorem KPI). Here is an example using `T` from above, with a choice of a codomain element that has a nonempty pre-image.

```
TK = T.kernel()
v = vector(QQ, [2, 3, 0, 1, -1])
u = T.preimage_representative(v)
u
```

```
(-1, 1, 0)
```

Now the following will create random elements of the preimage of `v`, which can be verified by the test always returning `True`. Use the compute cell just below if you are curious what `p` looks like.

```
p = u + TK.random_element()
T(p) == v
```

```
True
```

```
p                 # random
```

```
(-13/10, 23/20, 3/20)
```

As suggested, some choices of `v` can lead to empty pre-images, in which case Theorem KPI does not even apply.

```
v = vector(QQ, [4, 6, 3, 1, -2])
u = T.preimage_representative(v)
```

```
Traceback (most recent call last):
...
ValueError: element is not in the image
```

The situation is less interesting for an injective linear transformation. Still, pre-images may be empty, but when they are nonempty, they are just singletons (a single element) since the kernel is empty. So a repeat of the above example, with `S` rather than `T`, would not be very informative.

CILT Composition of Injective Linear Transformations

One way to use Sage is to construct examples of theorems and verify the conclusions. Sometimes you will get this wrong: you might build an example that does not satisfy the hypotheses, or your example may not satisfy the conclusions. This may be because you are not using Sage properly, or because you do not understand a definition or a theorem, or in very limited cases you may have uncovered a bug in Sage (which is always the preferred explanation!). But in the process of trying to understand a discrepancy or unexpected result, you will learn much more, both about linear algebra and about Sage. And Sage is incredibly patient — it will stay up with you all night to help you through a rough patch.

Let us illustrate the above in the context of Theorem CILTI. The hypotheses indicate we need two injective linear transformations. Where will get two such linear transformations? Well, the contrapositive of Theorem ILTD tells us that if the dimension of the domain exceeds the dimension of the codomain, we will never be injective. So we should at a minimum avoid this scenario. We can build two linear transformations from matrices created randomly, and just hope that they lead to injective linear transformations. Here is an example of how we create examples like this. The random matrix has single-digit entries, and almost always will lead to an injective linear transformation, though we cannot be absolutely certain. Evaluate this cell repeatedly, to see how rarely the result is not injective.

```
E = random_matrix(ZZ, 3, 2, x=-9, y=9)
T = linear_transformation(QQ^2, QQ^3, E, side='right')
T.is_injective()                                # random
```

```
True
```

Our concrete example below was created this way, so here we go.

```
U = QQ^2
V = QQ^3
W = QQ^4
A = matrix(QQ, 3, 2, [[-4, -1],
                      [-3,  3],
                      [ 7, -6]])
B = matrix(QQ, 4, 3, [[ 7,  0, -1],
                      [ 6,  2,  7],
                      [ 3, -1,  2],
                      [-6, -1,  1]])
T = linear_transformation(U, V, A, side='right')
T.is_injective()
```

```
True
```

```
S = linear_transformation(V, W, B, side='right')
S.is_injective()
```

```
True
```

```
C = S*T
C.is_injective()
```

```
True
```

Section SLT: Surjective Linear Transformations

SLT Surjective Linear Transformations

The situation in Sage for surjective linear transformations is similar to that for injective linear transformations. One distinction — what your text calls the range of a linear transformation is called the image of a transformation, obtained with the `.image()` method. Sage's term is more commonly used, so you are likely to see it again. As before, two examples, first up is Example RAO.

```
U = QQ^3
V = QQ^5
x1, x2, x3 = var('x1, x2, x3')
outputs = [ -x1 +   x2 - 3*x3,
            -x1 + 2*x2 - 4*x3,
             x1 +   x2 +   x3,
           2*x1 + 3*x2 +   x3,
             x1        + 2*x3]
T_symbolic(x1, x2, x3) = outputs
T = linear_transformation(U, V, T_symbolic)
T.is_surjective()
```

```
False
```

```
T.image()
```

```
Vector space of degree 5 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -3 -7 -2]
[ 0  1  2  5  1]
```

Besides showing the relevant commands in action, this demonstrates one half of Theorem RSLT. Now a reprise of Example FRAN.

```
U = QQ^5
V = QQ^3
x1, x2, x3, x4, x5 = var('x1, x2, x3, x4, x5')
outputs = [2*x1 +   x2 + 3*x3 - 4*x4 + 5*x5,
            x1 - 2*x2 + 3*x3 - 9*x4 + 3*x5,
           3*x1        + 4*x3 - 6*x4 + 5*x5]
S_symbolic(x1, x2, x3, x4, x5) = outputs
S = linear_transformation(U, V, S_symbolic)
S.is_surjective()
```

```
True
```

```
S.image()
```

```
Vector space of degree 3 and dimension 3 over Rational Field
Basis matrix:
[1 0 0]
[0 1 0]
[0 0 1]
```

```
S.image() == V
```

```
True
```

Previously, we have chosen elements of the codomain which have nonempty or empty preimages. We can now explain how to do this predictably. Theorem RPI explains that elements of the codomain with nonempty pre-images are precisely elements of the image. Consider the non-surjective linear transformation `T` from above.

```
TI = T.image()
B = TI.basis()
B
```

```
[
(1, 0, -3, -7, -2),
(0, 1, 2, 5, 1)
]
```

```
b0 = B[0]
b1 = B[1]
```

Now any linear combination of the basis vectors `b0` and `b1` must be an element of the image. Moreover, the first two slots of the resulting vector will equal the two scalars we choose to create the linear combination. But most importantly, see that the three remaining slots will be uniquely determined by these two choices. This means there is exactly one vector in the image with these values in the first two slots. So if we construct a new vector with these two values in the first two slots, and make any part of the last three slots even slightly different, we will form a vector that cannot be in the image, and will thus have a preimage that is an empty set. Whew, that is probably worth reading carefully several times, perhaps in conjunction with the example following.

```
in_image = 4*b0 + (-5)*b1
in_image
```

```
(4, -5, -22, -53, -13)
```

```
T.preimage_representative(in_image)
```

```
(-13, -9, 0)
```

```
outside_image = 4*b0 + (-5)*b1 + vector(QQ, [0, 0, 0, 0, 1])
outside_image
```

```
(4, -5, -22, -53, -12)
```

```
T.preimage_representative(outside_image)
```

```
Traceback (most recent call last):
...
ValueError: element is not in the image
```

CSLT Composition of Surjective Linear Transformations

As we mentioned in the last section, experimenting with Sage is a worthwhile complement to other methods of learning mathematics. We have purposely avoided providing illustrations of deeper results, such as Theorem ILTB and Theorem SLTB, which you should now be equipped to investigate yourself. For completeness, and since composition will be very important in the next few sections, we will provide an illustration of Theorem CSLTS. Similar to what we did in the previous section, we choose dimensions suggested by Theorem SLTD, and then use randomly constructed matrices to form a pair of surjective linear transformations.

```
U = QQ^4
V = QQ^3
W = QQ^2
A = matrix(QQ, 3, 4, [[ 3, -2,  8, -9],
                      [-1,  3, -4, -1],
                      [ 3,  2,  8,  3]])
T = linear_transformation(U, V, A, side='right')
```

```
T.is_surjective()
```

```
True
```

```
B = matrix(QQ, 2, 3, [[ 8, -5, 3],
                      [-2,  1, 1]])
S = linear_transformation(V, W, B, side='right')
S.is_surjective()
```

```
True
```

```
C = S*T
C.is_surjective()
```

```
True
```

Section IVLT: Invertible Linear Transformations

IVLT Invertible Linear Transformations

Of course, Sage can compute the inverse of a linear transformation. However, not every linear transformation has an inverse, and we will see shortly how to determine this. For now, take this example as just an illustration of the basics (both mathematically and for Sage).

```
U = QQ^4
V = QQ^4
x1, x2, x3, x4 = var('x1, x2, x3, x4')
outputs = [   x1 + 2*x2 - 5*x3 - 7*x4,
                     x2 - 3*x3 - 5*x4,
              x1 + 2*x2 - 4*x3 - 6*x4,
           -2*x1 - 2*x2 + 7*x3 + 8*x4 ]
T_symbolic(x1, x2, x3, x4) = outputs
T = linear_transformation(U, V, T_symbolic)
S = T.inverse()
S
```

```
Vector space morphism represented by the matrix:
[-8  7 -6  5]
[ 2 -3  2 -2]
[ 5 -3  4 -3]
[-2  2 -1  1]
Domain: Vector space of dimension 4 over Rational Field
Codomain: Vector space of dimension 4 over Rational Field
```

We can build the composition of `T` and its inverse, `S`, in both orders. We will optimistically name these as identity linear transformations, as predicted by Definition IVLT. Run the cells to define the compositions, then run the compute cells with the random vectors repeatedly — they should always return `True`.

```
IU = S*T
IV = T*S
```

```
u = random_vector(QQ, 4)
IU(u) == u      # random
```

```
True
```

```
v = random_vector(QQ, 4)
IV(v) == v      # random
```

```
True
```

We can also check that the compositions are the same as the identity linear transformation itself. We will do one, you can try the other.

```
id = linear_transformation(U, U, identity_matrix(QQ, 4))
IU.is_equal_function(id)
```

```
True
```

CIVLT Computing the Inverse of a Linear Transformations

Theorem ILTIS gives us a straightforward condition equivalence for an invertible linear transformation, but of course, it is even easier in Sage.

```
U = QQ^4
V = QQ^4
x1, x2, x3, x4 = var('x1, x2, x3, x4')
outputs = [   x1 + 2*x2 - 5*x3 - 7*x4,
                    x2 - 3*x3 - 5*x4,
              x1 + 2*x2 - 4*x3 - 6*x4,
           -2*x1 - 2*x2 + 7*x3 + 8*x4 ]
T_symbolic(x1, x2, x3, x4) = outputs
T = linear_transformation(U, V, T_symbolic)
T.is_invertible()
```

```
True
```

As easy as that is, it is still instructive to walk through an example similar to Example CIVLT using Sage, as a further illustration of the second half of the proof of Theorem ILTIS. Since T is surjective, every element of the codomain has a nonempty pre-image, and since T is injective, the pre-image of each element is a single element. Keep these facts in mind and convince yourself that the procedure below would never raise an error, and always has a unique result.

We first compute the pre-image of each element of a basis of the codomain.

```
preimages = [T.preimage_representative(v) for v in V.basis()]
preimages
```

```
[(-8, 7, -6, 5), (2, -3, 2, -2), (5, -3, 4, -3), (-2, 2, -1, 1)]
```

Then we define a new linear transformation, from `V` to `U`, which turns it around and uses the preimages as a set of images defining the new linear transformation. Explain to yourself how we know that `preimages` is a basis for `U`, and why this will create an invertible linear transformation.

```
S = linear_transformation(V, U, preimages)
S
```

```
Vector space morphism represented by the matrix:
[-8  7 -6  5]
[ 2 -3  2 -2]
[ 5 -3  4 -3]
[-2  2 -1  1]
Domain: Vector space of dimension 4 over Rational Field
Codomain: Vector space of dimension 4 over Rational Field
```

```
S.is_equal_function(T.inverse())
```

```
True
```

While this is a simple two-step procedure (form preimages, construct linear transformation), realize that this is *not* the process that Sage uses internally.

Notice that the essence of this construction is that when we work with an invertible linear transformation, the method `.preimage_representative()` behaves as a function (we mean the precise mathematical definition here) — it is always defined and always produces just one well-defined output. Here the `linear_transformation()` constructor is extending it to a linear function based on its action on a (finite) basis of the domain.

LTOE Linear Transformation Odds and Ends

We should mention that the notation `T^-1` will yield an inverse of a linear transformation in Sage.

```
U = QQ^4
V = QQ^4
x1, x2, x3, x4 = var('x1, x2, x3, x4')
outputs = [   x1 + 2*x2 - 5*x3 - 7*x4,
                     x2 - 3*x3 - 5*x4,
              x1 + 2*x2 - 4*x3 - 6*x4,
           -2*x1 - 2*x2 + 7*x3 + 8*x4]
T_symbolic(x1, x2, x3, x4) = outputs
T = linear_transformation(U, V, T_symbolic)
T^-1
```

```
Vector space morphism represented by the matrix:
[-8  7 -6  5]
[ 2 -3  2 -2]
[ 5 -3  4 -3]
[-2  2 -1  1]
Domain: Vector space of dimension 4 over Rational Field
Codomain: Vector space of dimension 4 over Rational Field
```

Also, the rank and nullity are what you might expect. Recall that for a matrix Sage provides a left nullity and a right nullity. There is no such distinction for linear transformations. We verify Theorem RPNDD as an example.

```
U = QQ^3
V = QQ^4
x1, x2, x3 = var('x1, x2, x3')
outputs = [ -x1        + 2*x3,
```

```
             x1 + 3*x2 + 7*x3,
             x1 +   x2 +   x3,
           2*x1 + 3*x2 + 5*x3]
R_symbolic(x1, x2, x3) = outputs
R = linear_transformation(QQ^3, QQ^4, R_symbolic)
R.rank()
```

```
2
```

```
R.nullity()
```

```
1
```

```
R.rank() + R.nullity() == U.dimension()
```

```
True
```

SUTH1 Sage Under The Hood, Round 1

We can parallel the above discussion about systems of linear equations and linear transformations using Sage. We begin with a matrix that we will use as a coefficient matrix for systems of equations, and then use the same matrix to define the associated linear transformation (acting on vectors placed to the right of the matrix).

```
A = matrix(QQ, [[-1, 0, 2],
                [ 1, 3, 7],
                [ 1, 1, 1],
                [ 2, 3, 5]])
T = linear_transformation(QQ^3, QQ^4, A, side='right')
```

We solve a linear system using the coefficient matrix, and compute an element of the pre-image of the linear transformation.

```
v = vector(QQ, [1, 2, 0, 1])
A.solve_right(v)
```

```
(-1, 1, 0)
```

```
T.preimage_representative(v)
```

```
(-1, 1, 0)
```

We compute a null space of the coefficient matrix and a kernel of the linear transformation, so as to understand the full solution set or the full preimage set.

```
A.right_kernel()
```

```
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[   1 -3/2  1/2]
```

```
T.kernel()
```

```
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[   1 -3/2  1/2]
```

We compute a column space of the coefficient matrix and an image (range) of the linear transformation to help us build an inconsistent system.

```
A.column_space()
```

```
Vector space of degree 4 and dimension 2 over Rational Field
Basis matrix:
[   1    0 -2/3   -1]
[   0    1  1/3    1]
```

```
T.image()
```

```
Vector space of degree 4 and dimension 2 over Rational Field
Basis matrix:
[   1    0 -2/3   -1]
[   0    1  1/3    1]
```

A vector that creates a system with no solution is a vector that has an empty pre-image.

```
v = vector(QQ, [1, 2, 1, 1])
A.solve_right(v)
```

```
Traceback (most recent call last):
...
ValueError: matrix equation has no solutions
```

```
T.preimage_representative(v)
```

```
Traceback (most recent call last):
...
ValueError: element is not in the image
```

Note that we could redo the above, replacing uses of "right" by "left" and uses of "column" by "row." The output would not change.

We suggest in the text that one could develop the theory of linear transformations from scratch, and then obtain many of our initial results about systems of equations and matrices as a consequence. In Sage it is the reverse. Sage implements many advanced concepts from various areas of mathematics by translating fundamental computations into the language of linear algebra. In turn, many of Sage's linear algebra routines ultimately depend on very fast algorithms for basic operations on matrices, such as computing an echelon form, a null space, or a span.

Chapter R
Representations

Section VR: Vector Representations

VR Vector Representations

Vector representation is described in the text in a fairly abstract fashion. Sage will support this view (which will be useful in the next section), as well as provide a more practical approach. We will explain both approaches. We begin with an arbitrarily chosen basis. We then create an alternate version of `QQ^4` with this basis as a"user basis", namely `V`.

```
v0 = vector(QQ, [ 1, 1, 1, 0])
v1 = vector(QQ, [ 1, 2, 3, 2])
v2 = vector(QQ, [ 2, 2, 3, 2])
v3 = vector(QQ, [-1, 3, 5, 5])
B = [v0, v1, v2, v3]
V = (QQ^4).subspace_with_basis(B)
V
```

```
Vector space of degree 4 and dimension 4 over Rational Field
User basis matrix:
[ 1  1  1  0]
[ 1  2  3  2]
[ 2  2  3  2]
[-1  3  5  5]
```

```
V.echelonized_basis_matrix()
```

```
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
```

Now, the construction of a linear transformation will use the basis provided for `V`. In the proof of Theorem VRLT we defined a linear transformation T that equaled ρ_B. T was defined by taking the basis vectors of B to the basis composed of standard unit vectors (Definition SUV). This is exactly what we will accomplish in the following construction. Note how the basis associated with the domain is automatically paired with the elements of the basis for the codomain.

```
rho = linear_transformation(V, QQ^4, (QQ^4).basis())
rho
```

```
Vector space morphism represented by the matrix:
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
Domain: Vector space of degree 4 and dimension 4 over Rational Field
User basis matrix:
[ 1  1  1  0]
[ 1  2  3  2]
[ 2  2  3  2]
[-1  3  5  5]
Codomain: Vector space of dimension 4 over Rational Field
```

First, we verify Theorem VRILT:

```
rho.is_invertible()
```

```
True
```

Notice that the matrix of the linear transformation is the identity matrix. This might look odd now, but we will have a full explanation soon. Let us see if this linear transformation behaves as it should. We will "coordinatize" an arbitrary vector, `w`.

```
w = vector(QQ, [-13, 28, 45, 43])
rho(w)
```

```
(3, 5, -6, 9)
```

```
lincombo = 3*v0 + 5*v1 + (-6)*v2 + 9*v3
lincombo
```

```
(-13, 28, 45, 43)
```

```
lincombo == w
```

```
True
```

Notice how the expression for `lincombo` is exactly the messy expression displayed in Definition VR. More precisely, we could even write this as:

```
w == sum([rho(w)[i]*B[i] for i in range(4)])
```

```
True
```

Or we can test this equality repeatedly with random vectors.

```
u = random_vector(QQ, 4)
u == sum([rho(u)[i]*B[i] for i in range(4)])
```

```
True
```

Finding a vector representation is such a fundamental operation that Sage has an easier command, bypassing the need to create a linear transformation. It does still require constructing a vector space with the alternate basis. Here goes, repeating the prior example.

```
w = vector(QQ, [-13, 28, 45, 43])
V.coordinate_vector(w)
```

```
(3, 5, -6, 9)
```

Boom!

SUTH2 Sage Under The Hood, Round 2

You will have noticed that we have never constructed examples involving our favorite abstract vector spaces, such as the vector space of polynomials with fixed maximum degree, the vector space of matrices of a fixed size, or even the crazy vector space. There is nothing to stop us (or you) from implementing these examples in Sage as vector spaces. Maybe someday it will happen. But since Sage is built to be a tool for serious mathematical research, the designers recognize that this is not necessary.

Theorem CFDVS tells us that *every* finite-dimensional vector space can be described (loosely speaking) by just a field of scalars (for us, $\mathbb{C}$ in the text, `QQ` in Sage) and the dimension. You can study whatever whacky vector space you might dream up, or whatever very complicated vector space that is important for particle physics, and through vector representation ("coordinatization"), you can convert your calculations to and from Sage.

Section MR: Matrix Representations

MR Matrix Representations

It is very easy to create matrix representations of linear transformations in Sage. Our examples in the text have used abstract vector spaces to make it a little easier to keep track of the domain and codomain. With Sage we will have to consistently use variants of `QQ^n`, but we will use non-obvious bases to make it nontrivial and interesting. Here are the components of our first example, one linear function, two vector spaces, four bases.

```
x1, x2, x3, x4 = var('x1, x2, x3, x4')
outputs = [3*x1 + 7*x2 + x3 - 4*x4,
           2*x1 + 5*x2 + x3 - 3*x4,
            -x1 - 2*x2      +   x4]
T_symbolic(x1, x2, x3, x4) = outputs
U = QQ^4
V = QQ^3
b0 = vector(QQ, [ 1, 1, -1, 0])
b1 = vector(QQ, [-1, 0, -2, 7])
b2 = vector(QQ, [ 0, 1, -2, 4])
b3 = vector(QQ, [-2, 0, -1, 6])
B = [b0, b1, b2, b3]
c0 = vector(QQ, [ 1,  6, -6])
c1 = vector(QQ, [ 0,  1, -1])
c2 = vector(QQ, [-2, -3,  4])
C = [c0, c1, c2]
d0 = vector(QQ, [ 1, -3,  2, -1])
d1 = vector(QQ, [ 0,  1,  0,  1])
d2 = vector(QQ, [-1,  2, -1, -1])
d3 = vector(QQ, [ 2, -8,  4, -3])
D = [d0, d1, d2, d3]
e0 = vector(QQ, [ 0,  1, -3])
e1 = vector(QQ, [-1,  2, -1])
```

```
e2 = vector(QQ, [ 2, -4,  3])
E = [e0, e1, e2]
```

We create alternate versions of the domain and codomain by providing them with bases other than the standard ones. Then we build the linear transformation and ask for its `.matrix()`. We will use numerals to distinguish our two examples.

```
U1 = U.subspace_with_basis(B)
V1 = V.subspace_with_basis(C)
T1 = linear_transformation(U1, V1, T_symbolic)
T1.matrix(side='right')
```

```
[ 15 -67 -25 -61]
[-75 326 120 298]
[  3 -17  -7 -15]
```

Now, we do it again, but with the bases `D` and `E`.

```
U2 = U.subspace_with_basis(D)
V2 = V.subspace_with_basis(E)
T2 = linear_transformation(U2, V2, T_symbolic)
T2.matrix(side='right')
```

```
[ -32    8   38  -91]
[-148   37  178 -422]
[ -80   20   96 -228]
```

So once the pieces are in place, it is quite easy to obtain a matrix representation. You might experiment with examples from the text, by first mentally converting elements of abstract vector spaces into column vectors via vector representations using simple bases for the abstract spaces.

The linear transformations `T1` and `T2` are not different as functions, despite the fact that Sage treats them as unequal (since they have different matrix representations). We can check that the two functions behave identically, first with random testing. Repeated execution of the following compute cell should always produce `True`.

```
u = random_vector(QQ, 4)
T1(u) == T2(u)
```

```
True
```

A better approach would be to see if `T1` and `T2` agree on a basis for the domain, and to avoid any favoritism, we will use the basis of `U` itself. Convince yourself that this is a proper application of Theorem LTDB to demonstrate equality.

```
all([T1(u) == T2(u) for u in U.basis()])
```

```
True
```

Or we can just ask if they are equal functions (a method that is implemented using the previous idea about agreeing on a basis).

```
T1.is_equal_function(T2)
```

```
True
```

We can also demonstrate Theorem FTMR — we will use the representation for `T1`. We need a couple of vector representation linear transformations.

```
rhoB = linear_transformation(U1, U, U.basis())
rhoC = linear_transformation(V1, V, V.basis())
```

Now we experiment with a "random" element of the domain. Study the third computation carefully, as it is the Sage version of Theorem FTMR. You might compute some of the pieces of this expression to build up the final expression, and you might duplicate the experiment with a different input and/or with the `T2` version.

```
T_symbolic(-3, 4, -5, 2)
```

```
(6, 3, -3)
```

```
u = vector(QQ, [-3, 4, -5, 2])
T1(u)
```

```
(6, 3, -3)
```

```
(rhoC^-1)(T1.matrix(side='right')*rhoB(u))
```

```
(6, 3, -3)
```

One more time, but we will replace the vector representation linear transformations with Sage conveniences. Recognize that the `.linear_combination_of_basis()` method is the inverse of vector representation (it is "un-coordinatization").

```
output_coord = T1.matrix(side='right')*U1.coordinate_vector(u)
V1.linear_combination_of_basis(output_coord)
```

```
(6, 3, -3)
```

We have concentrated here on the first version of the conclusion of Theorem FTMR. You could experiment with the second version in a similar manner. Extra credit: what changes do you need to make in any of these experiments if you remove the `side='right'` option?

SUTH3 Sage Under The Hood, Round 3

We have seen that Sage is able to add two linear transformations, or multiply a single linear transformation by a scalar. Under the hood, this is accomplished by simply adding matrix representations, or multiplying a matrix by a scalar, according to Theorem MRSLT and Theorem MRMLT (respectively). Theorem MRCLT says linear transformation composition is matrix representation multiplication. Is it still a mystery why we use the symbol `*` for linear transformation composition in Sage?

We could do several examples here, but you should now be able to construct them yourselves. We will do just one, linear transformation composition is matrix representation multiplication.

```
x1, x2, x3, x4 = var('x1, x2, x3, x4')
outputs = [-5*x1 - 2*x2 +   x3,
            4*x1 - 3*x2 - 3*x3,
            4*x1 - 6*x2 - 4*x3,
            5*x1 + 3*x2       ]
T_symbolic(x1, x2, x3) = outputs
outputs = [-3*x1 - x2 + x3 + 2*x4,
            7*x1 + x2 + x3 -   x4]
S_symbolic(x1, x2, x3, x4) = outputs
b0 = vector(QQ, [-1, -2,  2])
b1 = vector(QQ, [ 1,  1,  0])
b2 = vector(QQ, [ 0,  3, -5])
U = (QQ^3).subspace_with_basis([b0, b1, b2])
c0 = vector(QQ, [ 0,  0,  2,  1])
c1 = vector(QQ, [ 2, -3, -1, -6])
c2 = vector(QQ, [-2,  3,  2,  7])
c3 = vector(QQ, [ 1, -2, -4, -6])
V = (QQ^4).subspace_with_basis([c0, c1, c2, c3])
d0 = vector(QQ, [3, 4])
d1 = vector(QQ, [2, 3])
W = (QQ^2).subspace_with_basis([d0, d1])
T = linear_transformation(U, V, T_symbolic)
S = linear_transformation(V, W, S_symbolic)
(S*T).matrix('right')
```

```
[-321  218  297]
[ 456 -310 -422]
```

```
S.matrix(side='right')*T.matrix(side='right')
```

```
[-321  218  297]
[ 456 -310 -422]
```

Extra credit: what changes do you need to make if you dropped the `side='right'` option on these three matrix representations?

LTR Linear Transformation Restrictions

Theorem KNSI and Theorem RCSI have two of the most subtle proofs we have seen so far. The conclusion that two vector spaces are isomorphic is established by actually constructing an isomorphism between the vector spaces. To build the isomorphism, we begin with a familar object, a vector representation linear transformation, but the hard work is showing that we can "restrict" the domain and codomain of this function and still arrive at a legitimate (invertible) linear transformation. In an effort to make the proofs more concrete, we will walk through a nontrivial example for Theorem KNSI, and you might try to do the same for Theorem RCSI. (An understanding of this subsection is not needed for the remainder — its second purpose is to demonstrate some of the powerful tools Sage provides.)

Here are the pieces. We build a linear transformation with two different representations, one with respect to standard bases, the other with respect to less-obvious bases.

```
x1, x2, x3, x4, x5 = var('x1, x2, x3, x4, x5')
outputs = [ x1 - x2 - 5*x3 +   x4 +   x5,
            x1      - 2*x3 -   x4 -   x5,
               - x2 - 3*x3 + 2*x4 + 2*x5,
```

```
           -x1 + x2 + 5*x3 -   x4 -   x5]
T_symbolic(x1, x2, x3, x4, x5) = outputs
b0 = vector(QQ, [-1,  6,  5,  5,  1])
b1 = vector(QQ, [-1,  5,  4,  4,  1])
b2 = vector(QQ, [-2,  4,  3,  2,  5])
b3 = vector(QQ, [ 1, -1,  0,  1, -5])
b4 = vector(QQ, [ 3, -7, -6, -5, -4])
U = (QQ^5).subspace_with_basis([b0, b1, b2, b3, b4])
c0 = vector(QQ, [1, 1, 1, -3])
c1 = vector(QQ, [-2, 3, -6, -7])
c2 = vector(QQ, [0, -1, 1, 2])
c3 = vector(QQ, [-1, 3, -4, -7])
V = (QQ^4).subspace_with_basis([c0, c1, c2, c3])
T_plain = linear_transformation(QQ^5, QQ^4, T_symbolic)
T_fancy = linear_transformation(   U,    V, T_symbolic)
```

Now we compute the kernel of the linear transformation using the "plain" version, and the null space of a matrix representation coming from the "fancy" version.

```
K = T_plain.kernel()
K
```

```
Vector space of degree 5 and dimension 3 over Rational Field
Basis matrix:
[   1    0  2/7    0  3/7]
[   0    1 -1/7    0  2/7]
[   0    0    0    1   -1]
```

```
MK = T_fancy.matrix(side='right').right_kernel()
MK
```

```
Vector space of degree 5 and dimension 3 over Rational Field
Basis matrix:
[      1       0       0 -97/203 164/203]
[      0       1       0  -10/29   19/29]
[      0       0       1 129/203 100/203]
```

So quite obviously, the kernel of the linear transformation is quite different looking from the null space of the matrix representation. Though it is no accident that they have the same dimension. Now we build the necessary vector representation, and use two Sage commands to "restrict" the function to a smaller domain (the kernel of the linear transformation) and a smaller codomain (the null space of the matrix representation relative to nonstandard bases).

```
rhoB = linear_transformation(U, QQ^5, (QQ^5).basis())
rho_restrict = rhoB.restrict_domain(K).restrict_codomain(MK)
rho_restrict
```

```
Vector space morphism represented by the matrix:
[ 33/7 -37/7 -11/7]
[-13/7  22/7 -12/7]
[   -4     5     6]
Domain: Vector space of degree 5 and dimension 3 over Rational Field
Basis matrix:
[   1    0  2/7    0  3/7]
[   0    1 -1/7    0  2/7]
```

```
[   0    0    0    1   -1]
Codomain: Vector space of degree 5 and dimension 3 over Rational Field
Basis matrix:
[      1       0       0 -97/203 164/203]
[      0       1       0  -10/29   19/29]
[      0       0       1 129/203 100/203]
```

The first success is that the restriction was even created. Sage would recognize if the original linear transformation ever carried an input from the restricted domain to an output that was not contained in the proposed codomain, and would have raised an error in that event. Phew! Guaranteeing this success was the first big step in the proof of Theorem KNSI. Notice that the matrix representation of the restriction is a 3×3 matrix, since the restriction runs between a domain and codomain that each have dimension 3. These two vector spaces (the domain and codomain of the restriction) have dimension 3 but still contain vectors with 5 entries in their un-coordinatized versions.

The next two steps of the proof show that the restriction is injective (easy in the proof) and surjective (hard in the proof). In Sage, here is the second success,

```
rho_restrict.is_injective()
```

```
True
```

```
rho_restrict.is_surjective()
```

```
True
```

Verified as invertible, `rho_restrict` qualifies as an isomorphism between the linear transformation kernel, `K`, and the matrix representation null space, `MK`. Only an example, but still very nice. Your turn — can you create a verfication of Theorem RCSI (for this example, or some other nontrivial example you might create yourself)?

NME9 Nonsingular Matrix Equivalences, Round 9

Our final fact about nonsingular matrices expresses the correspondence between invertible matrices and invertible linear transformations. As a Sage demonstration, we will begin with an invertible linear transformation and examine two matrix representations. We will create the linear transformation with nonstandard bases and then compute its representation relative to standard bases.

```
x1, x2, x3, x4 = var('x1, x2, x3, x4')
outputs = [ x1        - 2*x3 - 4*x4,
                  x2 -   x3 - 5*x4,
           -x1 - 2*x2 + 2*x3 + 7*x4,
               -   x2        +   x4]
T_symbolic(x1, x2, x3, x4) = outputs
b0 = vector(QQ, [ 1, -2, -1,  8])
b1 = vector(QQ, [ 0,  1,  0, -2])
b2 = vector(QQ, [-1, -2,  2, -5])
b3 = vector(QQ, [-1, -3,  2, -2])
U = (QQ^4).subspace_with_basis([b0, b1, b2, b3])
c0 = vector(QQ, [ 3, -1,  4, -8])
c1 = vector(QQ, [ 1,  0,  1, -1])
c2 = vector(QQ, [ 0,  2, -1,  6])
c3 = vector(QQ, [-1,  2, -2,  8])
```

```
V = (QQ^4).subspace_with_basis([c0, c1, c2, c3])
T = linear_transformation(U, V, T_symbolic)
T
```

```
Vector space morphism represented by the matrix:
[ 131  -56 -321  366]
[ -37   17   89 -102]
[ -61   25  153 -173]
[  -7   -1   24  -25]
Domain: Vector space of degree 4 and dimension 4 over Rational Field
User basis matrix:
[ 1 -2 -1  8]
[ 0  1  0 -2]
[-1 -2  2 -5]
[-1 -3  2 -2]
Codomain: Vector space of degree 4 and dimension 4 over Rational Field
User basis matrix:
[ 3 -1  4 -8]
[ 1  0  1 -1]
[ 0  2 -1  6]
[-1  2 -2  8]
```

```
T.is_invertible()
```

```
True
```

```
T.matrix(side='right').is_invertible()
```

```
True
```

```
(T^-1).matrix(side='right') == (T.matrix(side='right'))^-1
```

```
True
```

We can convert `T` to a new representation using standard bases for `QQ^4` by computing images of the standard basis.

```
images = [T(u) for u in (QQ^4).basis()]
T_standard = linear_transformation(QQ^4, QQ^4, images)
T_standard
```

```
Vector space morphism represented by the matrix:
[ 1  0 -1  0]
[ 0  1 -2 -1]
[-2 -1  2  0]
[-4 -5  7  1]
Domain: Vector space of dimension 4 over Rational Field
Codomain: Vector space of dimension 4 over Rational Field
```

```
T_standard.matrix(side='right').is_invertible()
```

```
True
```

Understand that *any* matrix representation of `T_symbolic` will have an invertible matrix representation, no matter which bases are used. If you look at the matrix representation of `T_standard` and the definition of `T_symbolic` the construction of this example will be transparent, especially if you know the random matrix constructor,

```
A = random_matrix(QQ, 4, algorithm='unimodular', upper_bound=9)
A                                                          # random
```

```
[-1 -1  2  1]
[ 1  1 -1  0]
[-2 -1  5  6]
[ 1  0 -4 -5]
```

Section CB: Change of Basis

ENDO Endomorphisms

An **endomorphism** is an "operation-preserving" function (a "morphism") whose domain and codomain are equal. Sage takes this definition one step further for linear transformations and requires that the domain and codomain have the same bases (either a default echelonized basis or the same user basis). When a linear transformation meets this extra requirement, several natural methods become available.

Principally, we can compute the eigenvalues provided by Definition EELT. We also get a natural notion of a characteristic polynomial.

```
x1, x2, x3, x4 = var('x1, x2, x3, x4')
outputs = [ 4*x1 + 2*x2 -   x3 + 8*x4,
            3*x1 - 5*x2 - 9*x3       ,
                   6*x2 + 7*x3 + 6*x4,
           -3*x1 + 2*x2 + 5*x3 - 3*x4]
T_symbolic(x1, x2, x3, x4) = outputs
T = linear_transformation(QQ^4, QQ^4, T_symbolic)
T.eigenvalues()
```

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

```
cp = T.characteristic_polynomial()
cp
```

```
x^4 - 3*x^3 - 3*x^2 + 11*x - 6
```

```
cp.factor()
```

```
(x - 3) * (x + 2) * (x - 1)^2
```

Now the question of eigenvalues being elements of the set of scalars used for the vector space becomes even more obvious. If we define an endomorphism on a vector space whose scalars are the rational numbers, should we "allow" irrational or complex eigenvalues? You will now recognize our use of the complex numbers in the text for the gross convenience that it is.

CBM Change-of-Basis Matrix

To create a change-of-basis matrix, it is enough to construct an identity linear transformation relative to a domain and codomain with the specified user bases, which is simply a straight application of Definition CBM. Here we go with two arbitrary bases.

```
b0 = vector(QQ, [-5, 8,  0, 4])
b1 = vector(QQ, [-3, 9, -2, 4])
b2 = vector(QQ, [-1, 4, -1, 2])
b3 = vector(QQ, [-1, 2,  0, 1])
B = [b0, b1, b2, b3]
U = (QQ^4).subspace_with_basis(B)
c0 = vector(QQ, [ 0,  2, -7,  5])
c1 = vector(QQ, [-1,  2, -1,  4])
c2 = vector(QQ, [ 1, -3,  5, -7])
c3 = vector(QQ, [ 1,  1, -8,  3])
C = [c0, c1, c2, c3]
V = (QQ^4).subspace_with_basis(C)
x1, x2, x3, x4 = var('x1, x2, x3, x4')
id_symbolic(x1, x2, x3, x4) = [x1, x2, x3, x4]
S = linear_transformation(U, V, id_symbolic)
CB = S.matrix(side='right')
CB
```

```
[ 36  25   8   7]
[ 27  34  15   7]
[ 35  35  14   8]
[-13  -4   0  -2]
```

```
S.is_invertible()
```

```
True
```

We can demonstrate that `CB` is indeed the change-of-basis matrix from `B` to `C`, converting vector representations relative to `B` into vector representations relative to `C`. We choose an arbitrary vector, `x`, to experiment with (you could experiment with other possibilities). We use the Sage conveniences to create vector representations relative to the two bases, and then verify Theorem CBM. Recognize that `x`, `u` and `v` are all the same vector.

```
x = vector(QQ, [-45, 62, 171, 85])
u = U.coordinate_vector(x)
u
```

```
(-103, -108, 45, 839)
```

```
v = V.coordinate_vector(x)
v
```

```
(-175, 95, -43, 93)
```

```
v == CB*u
```

```
True
```

We can also verify the construction above by building the change-of-basis matrix directly (i.e., without constructing a linear transformation).

```
cols = [V.coordinate_vector(u) for u in U.basis()]
M = column_matrix(cols)
M
```

```
[ 36  25   8   7]
[ 27  34  15   7]
[ 35  35  14   8]
[-13  -4   0  -2]
```

MRCB Matrix Representation and Change-of-Basis

In Sage MR we built two matrix representations of one linear transformation, relative to two different pairs of bases. We now understand how these two matrix representations are related — Theorem MRCB gives the precise relationship with change-of-basis matrices, one converting vector representations on the domain, the other converting vector representations on the codomain. Here is the demonstration. We use `MT` as the prefix of names for matrix representations, `CB` as the prefix for change-of-basis matrices, and numerals to distinguish the two domain-codomain pairs.

```
x1, x2, x3, x4 = var('x1, x2, x3, x4')
outputs = [3*x1 + 7*x2 + x3 - 4*x4,
           2*x1 + 5*x2 + x3 - 3*x4,
            -x1 - 2*x2      +   x4]
T_symbolic(x1, x2, x3, x4) = outputs
U = QQ^4
V = QQ^3
b0 = vector(QQ, [ 1, 1, -1, 0])
b1 = vector(QQ, [-1, 0, -2, 7])
b2 = vector(QQ, [ 0, 1, -2, 4])
b3 = vector(QQ, [-2, 0, -1, 6])
B = [b0, b1, b2, b3]
c0 = vector(QQ, [ 1,  6, -6])
c1 = vector(QQ, [ 0,  1, -1])
c2 = vector(QQ, [-2, -3,  4])
C = [c0, c1, c2]
d0 = vector(QQ, [ 1, -3,  2, -1])
d1 = vector(QQ, [ 0,  1,  0,  1])
d2 = vector(QQ, [-1,  2, -1, -1])
d3 = vector(QQ, [ 2, -8,  4, -3])
D = [d0, d1, d2, d3]
e0 = vector(QQ, [ 0,  1, -3])
e1 = vector(QQ, [-1,  2, -1])
e2 = vector(QQ, [ 2, -4,  3])
E = [e0, e1, e2]
U1 = U.subspace_with_basis(B)
V1 = V.subspace_with_basis(C)
T1 = linear_transformation(U1, V1, T_symbolic)
MTBC =T1.matrix(side='right')
MTBC
```

```
[ 15 -67 -25 -61]
[-75 326 120 298]
[  3 -17  -7 -15]
```

```
U2 = U.subspace_with_basis(D)
V2 = V.subspace_with_basis(E)
T2 = linear_transformation(U2, V2, T_symbolic)
MTDE = T2.matrix(side='right')
MTDE
```

```
[ -32    8   38  -91]
[-148   37  178 -422]
[ -80   20   96 -228]
```

This is as far as we could go back in Section MR. These two matrices represent the same linear transformation (namely `T_symbolic`), but the question now is "how are these representations related?" We need two change-of-basis matrices. Notice that with different dimensions for the domain and codomain, we get square matrices of different sizes.

```
identity4(x1, x2, x3, x4) = [x1, x2, x3, x4]
CU = linear_transformation(U2, U1, identity4)
CBDB = CU.matrix(side='right')
CBDB
```

```
[ 6  7 -8  1]
[ 5  1 -5  9]
[-9 -6 10 -9]
[ 0  3 -1 -5]
```

```
identity3(x1, x2, x3) = [x1, x2, x3]
CV = linear_transformation(V1, V2, identity3)
CBCE = CV.matrix(side='right')
CBCE
```

```
[  8   1  -7]
[ 33   4 -28]
[ 17   2 -15]
```

Finally, here is Theorem MRCB, relating the the two matrix representations via the change-of-basis matrices.

```
MTDE == CBCE * MTBC * CBDB
```

```
True
```

We can walk through this theorem just a bit more carefully, step-by-step. We will compute three matrix-vector products, using three vector representations, to demonstrate the equality above. To prepare, we choose the vector `x` arbitrarily, and we compute its value when evaluted by `T_symbolic`, and then verify the vector and matrix representations relative to `D` and `E`.

```
T_symbolic(34, -61, 55, 18)
```

```
(-342, -236, 106)
```

```
x = vector(QQ, [34, -61, 55, 18])
u_D = U2.coordinate_vector(x)
u_D
```

```
(25, 24, -13, -2)
```

```
v_E = V2.coordinate_vector(vector(QQ, [-342, -236, 106]))
v_E
```

```
(-920, -4282, -2312)
```

```
v_E == MTDE*u_D
```

```
True
```

So far this is not really new, we have just verified the representation `MTDE` in the case of one input vector (`x`), but now we will use the alternate version of this matrix representation, `CBCE * MTBC * CBDB`, in steps.

First, convert the input vector from a representation relative to `D` to a representation relative to `B`.

```
u_B = CBDB*u_D
u_B
```

```
(420, 196, -481, 95)
```

Now apply the matrix representation, which expects "input" coordinatized relative to `B` and produces "output" coordinatized relative to `C`.

```
v_C = MTBC*u_B
v_C
```

```
(-602, 2986, -130)
```

Now convert the output vector from a representation relative to `C` to a representation relative to `E`.

```
v_E = CBCE*v_C
v_E
```

```
(-920, -4282, -2312)
```

It is no surprise that this version of `v_E` equals the previous one, since we have checked the equality of the matrices earlier. But it may be instructive to see the input converted by change-of-basis matrices before and after being hit by the linear transformation (as a matrix representation). Now we will perform another example, but this time using Sage endomorphisms, linear transformations with equal bases for the domain and codomain. This will allow us to illustrate Theorem SCB. Just for fun, we will do something large. Notice the labor-saving device for manufacturing many symbolic variables at once.

```
[var('x{0}'.format(i)) for i in range(1, 12)]
```

```
[x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11]
```

```
x = vector(SR, [x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11])
A = matrix(QQ, 11, [
  [ 146, -225, -10,  212,  419, -123, -73,  3,  219, -100, -57],
  [ -24,   32,   1,  -33,  -66,   13,  16,  1,  -33,   18,   3],
  [  79, -131, -15,  124,  235,  -74, -33, -3,  128,  -57, -29],
  [  -1,   13, -16,   -1,  -27,    3,  -5, -4,   -9,    6,   2],
  [-104,  170,  20, -162, -307,   95,  45,  3, -167,   75,  37],
  [ -16,   59, -19,  -34, -103,   27, -10, -1,  -51,   27,   8],
  [  36,  -41,  -7,   46,   80,  -25, -26,  2,   42,  -18, -16],
  [  -5,    0,   1,   -4,   -3,    2,   6, -1,    0,   -2,   3],
  [ 105, -176, -28,  168,  310, -103, -41, -4,  172,  -73, -40],
  [   1,    7,   0,   -3,   -9,    5,  -6, -2,   -7,    3,   2],
  [  74, -141,   4,  118,  255,  -72, -23, -1,  133,  -63, -26]
                  ])
out = (A*x).list()
T_symbolic(x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11) = out
V1 = QQ^11
C = V1.basis()
T1 = linear_transformation(V1, V1, T_symbolic)
MC = T1.matrix(side='right')
MC
```

```
[ 146 -225  -10  212  419 -123  -73    3  219 -100  -57]
[ -24   32    1  -33  -66   13   16    1  -33   18    3]
[  79 -131  -15  124  235  -74  -33   -3  128  -57  -29]
[  -1   13  -16   -1  -27    3   -5   -4   -9    6    2]
[-104  170   20 -162 -307   95   45    3 -167   75   37]
[ -16   59  -19  -34 -103   27  -10   -1  -51   27    8]
[  36  -41   -7   46   80  -25  -26    2   42  -18  -16]
[  -5    0    1   -4   -3    2    6   -1    0   -2    3]
[ 105 -176  -28  168  310 -103  -41   -4  172  -73  -40]
[   1    7    0   -3   -9    5   -6   -2   -7    3    2]
[  74 -141    4  118  255  -72  -23   -1  133  -63  -26]
```

Not very interesting, and perhaps even transparent, with a definiton from a matrix and with the standard basis attached to `V1 == QQ^11`. Let us use a different basis to obtain a more interesting representation. We will input the basis compactly as the columns of a nonsingular matrix.

```
D = matrix(QQ, 11,
    [[ 1,  2, -1, -2,  4,  2,  2, -2,  4,  4,  8],
     [ 0,  1,  0,  2, -2,  1,  1, -1, -7,  5,  3],
     [ 1,  0,  0, -2,  3,  0, -1, -1,  6, -1, -1],
     [ 0, -1,  1, -1,  3, -2, -3,  0,  5, -8,  2],
     [-1,  0,  0,  3, -4,  0,  1,  1, -8,  1,  2],
     [-1, -1,  1,  0,  3, -3, -4, -1,  0, -7,  3],
     [ 0,  1,  0,  0,  2,  0,  0, -1,  0, -1,  8],
     [ 0,  0,  0, -1,  0,  0,  0,  1,  5, -4,  1],
     [ 1,  0,  0, -2,  3,  0, -2, -3,  3,  3, -4],
     [ 0, -1,  0,  0,  1, -1, -2, -1,  2, -4,  0],
     [ 1,  0, -1, -2,  0,  2,  2,  0,  5,  3, -1]])
E = D.columns()
```

```
V2 = (QQ^11).subspace_with_basis(E)
T2 = linear_transformation(V2, V2, T_symbolic)
MB = T2.matrix(side='right')
MB
```

```
[ 2  1  0  0  0  0  0  0  0  0  0]
[ 0  2  1  0  0  0  0  0  0  0  0]
[ 0  0  2  0  0  0  0  0  0  0  0]
[ 0  0  0  2  1  0  0  0  0  0  0]
[ 0  0  0  0  2  0  0  0  0  0  0]
[ 0  0  0  0  0 -1  1  0  0  0  0]
[ 0  0  0  0  0  0 -1  1  0  0  0]
[ 0  0  0  0  0  0  0 -1  1  0  0]
[ 0  0  0  0  0  0  0  0 -1  1  0]
[ 0  0  0  0  0  0  0  0  0 -1  1]
[ 0  0  0  0  0  0  0  0  0  0 -1]
```

Well, now *that* is interesting! What a nice representation. Of course, it is all due to the choice of the basis (which we have not explained). To explain the relationship between the two matrix representations, we need a change-of-basis-matrix, and its inverse. Theorem SCB says we need the matrix that converts vector representations relative to `B` into vector representations relative to `C`.

```
out = [x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11]
id11(x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11) = out
L = linear_transformation(V2, V1, id11)
CB = L.matrix(side='right')
CB
```

```
[ 1  2 -1 -2  4  2  2 -2  4  4  8]
[ 0  1  0  2 -2  1  1 -1 -7  5  3]
[ 1  0  0 -2  3  0 -1 -1  6 -1 -1]
[ 0 -1  1 -1  3 -2 -3  0  5 -8  2]
[-1  0  0  3 -4  0  1  1 -8  1  2]
[-1 -1  1  0  3 -3 -4 -1  0 -7  3]
[ 0  1  0  0  2  0  0 -1  0 -1  8]
[ 0  0  0 -1  0  0  0  1  5 -4  1]
[ 1  0  0 -2  3  0 -2 -3  3  3 -4]
[ 0 -1  0  0  1 -1 -2 -1  2 -4  0]
[ 1  0 -1 -2  0  2  2  0  5  3 -1]
```

OK, all set.

```
CB^-1*MC*CB
```

```
[ 2  1  0  0  0  0  0  0  0  0  0]
[ 0  2  1  0  0  0  0  0  0  0  0]
[ 0  0  2  0  0  0  0  0  0  0  0]
[ 0  0  0  2  1  0  0  0  0  0  0]
[ 0  0  0  0  2  0  0  0  0  0  0]
[ 0  0  0  0  0 -1  1  0  0  0  0]
[ 0  0  0  0  0  0 -1  1  0  0  0]
[ 0  0  0  0  0  0  0 -1  1  0  0]
[ 0  0  0  0  0  0  0  0 -1  1  0]
[ 0  0  0  0  0  0  0  0  0 -1  1]
[ 0  0  0  0  0  0  0  0  0  0 -1]
```

Which is `MB`. So the conversion from a "messy" matrix representation relative to a standard basis to a "clean" representation relative to some other basis is just a similarity transformation by a change-of-basis matrix. Oh, I almost forgot. Where did that basis come from? Hint: work your way up to Section JCF.

CELT Designing Matrix Representations

How do we find the eigenvectors of a linear transformation? How do we find pleasing (or computationally simple) matrix representations of linear transformations? Theorem EER and Theorem SCB applied in the context of Theorem DC can answer both questions. Here is an example.

```
x1, x2, x3, x4, x5, x6 = var('x1, x2, x3, x4,x5,x6')
outputs = [  9*x1 - 15*x2 - 7*x3 + 15*x4 - 36*x5 - 53*x6,
            24*x1 - 20*x2 - 9*x3 + 18*x4 - 24*x5 - 78*x6,
             8*x1 -  6*x2 - 3*x3 +  6*x4 -  6*x5 - 26*x6,
           -12*x1 -  9*x2 - 3*x3 + 13*x4 - 54*x5 - 24*x6,
            -8*x1 +  6*x2 + 3*x3 -  6*x4 +  6*x5 + 26*x6,
            -4*x1 -  3*x2 -   x3 +  3*x4 - 18*x5 -  4*x6]
T_symbolic(x1, x2, x3, x4, x5, x6) = outputs
T1 = linear_transformation(QQ^6, QQ^6, T_symbolic)
M1 = T1.matrix(side='right')
M1
```

```
[  9 -15  -7  15 -36 -53]
[ 24 -20  -9  18 -24 -78]
[  8  -6  -3   6  -6 -26]
[-12  -9  -3  13 -54 -24]
[ -8   6   3  -6   6  26]
[ -4  -3  -1   3 -18  -4]
```

Now we compute the eigenvalues and eigenvectors of `M1`. Since `M1` is diagonalizable, we can find a basis of eigenvectors for use as the basis for a new representation.

```
ev = M1.eigenvectors_right()
ev
```

```
[(4, [
(1, 6/5, 2/5, 4/5, -2/5, 1/5)
], 1), (0, [
(1, 9/7, 4/7, 3/7, -3/7, 1/7)
], 1), (-2, [
(1, 7/5, 2/5, 3/5, -2/5, 1/5)
], 1), (-3, [
(1, 3, 1, -3/2, -1, -1/2)
], 1), (1, [
(1, 0, 0, 3, 0, 1),
(0, 1, 1/3, -2, -1/3, -2/3)
], 2)]
```

```
evalues, evectors = M1.eigenmatrix_right()
B = evectors.columns()
V = (QQ^6).subspace_with_basis(B)
T2 = linear_transformation(V, V, T_symbolic)
```

```
M2 = T2.matrix('right')
M2
```

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

The eigenvectors that are the basis elements in B are the eigenvectors of the linear transformation, *relative* to the standard basis. For different representations the eigenvectors take different forms, relative to other bases. What are the eigenvectors of the matrix representation M2?

Notice that the eigenvalues of the linear transformation are totally independent of the representation. So in a sense, they are an inherent property of the linear transformation.

You should be able to use these techniques with linear transformations on abstract vector spaces — just use a mental linear transformation transforming the abstract vector space back-and-forth between a vector space of column vectors of the right size.

SUTH4 Sage Under The Hood, Round 4

We finally have enough theorems to understand how Sage creates and manages linear transformations. With a choice of bases for the domain and codomain, a linear transformation can be represented by a matrix. Every interesting property of the linear transformation can be computed from the matrix representation, and we can convert between representations (of vectors and linear transformations) with change-of-basis matrices, similarity and matrix multiplication.

So we can understand the theory of linear algebra better by experimenting with the assistance of Sage, and the theory of linear algebra helps us understand how Sage is designed and functions. A virtuous cycle, if there ever was one. Keep it going.