Lab Manual

for

Linear Algebra

by

Jim Hefferon

Cover: my Chocolate Lab, Suzy.

Preface

This collection supplements the text *Linear Algebra*[1] with explorations to help students solidify and extend their understanding of the subject, using the mathematical software *Sage*.[2]

A major goal of any undergraduate Mathematics program is to move students toward a higher-level, more abstract, grasp of the subject. For instance, Calculus classes work on elaborate computations while later courses spend more effort on concepts and proofs, focusing less on the details of calculations.

The text *Linear Algebra* fits into this development process. Its presentation works to bring students to a deeper understanding, but it does so by expecting that for them at this point a good bit of calculation helps the process. Naturally it uses examples and practice problems that are small-sized and have manageable numbers: an assignment to by-hand multiply a pair of three by three matrices of small integers will build intuition, whereas asking students to do that same question with twenty by twenty matrices of ten decimal place numbers would be badgering.

However, an instructor can worry that this misses a chance to make the point that Linear Algebra is widely applied, or to develop students's understanding through explorations that are not hindered by the mechanics of paper and pencil. Mathematical software can mitigate these concerns by extending the reach of what is reasonable to bigger systems, harder numbers, and longer computations. This manual's goal is to extends students's ability to do problems in that way. For instance, an advantage of learning how to automate work and handle larger jobs is that this is more like what a student must do when applying Linear Algebra to other areas. Another advantage is that students see new ideas such as runtime growth measures.

Well then, why not teach straight from the computer?

Our goal is to develop a higher-level understanding of the material so we keep the focus on vector spaces and linear maps. The exposition here takes computation to be a tool to develop that understanding, not the main object.

Some instructors will find that their students are best served by keeping a tight focus on the core material and leaving aside altogether the work in this manual. Other instructors have students who will benefit from the increased reach that the software provides. This manual's existence, and status as a separate book, gives teachers the freedom to make the choice that suits their class.

[1] The text's home page http://joshua.smcvt.edu/linearalgebra has the PDF, the ancillary materials, and the LaTeX source.
[2] See http://www.sagemath.org for the software and documentation.

Why Sage?

In *Open Source Mathematical Software* [Joyner and Stein, 2007][1] the authors argue that for Mathematics the best way forward is to use software that is Open Source.

> Suppose Jane is a well-known mathematician who announces she has proved a theorem. We probably will believe her, but she knows that she will be required to produce a proof if requested. However, suppose now Jane says a theorem is true based partly on the results of software. The closest we can reasonably hope to get to a rigorous proof (without new ideas) is the open inspection and ability to use all the computer code on which the result depends. If the program is proprietary, this is not possible. We have every right to be distrustful, not only due to a vague distrust of computers but because even the best programmers regularly make mistakes.
>
> If one reads the proof of Jane's theorem in hopes of extending her ideas or applying them in a new context, it is limiting to not have access to the inner workings of the software on which Jane's result builds.

Professionals choose their tools by balancing many factors but this argument is persuasive. We use *Sage* because it is both very capable so that students can learn a great deal from it, and because it is Free[2] and Open Source.[3]

This manual

This is Free. Get the latest version from http://joshua.smcvt.edu/linearalgebra. Also see that page for the license details and for the LaTeX source. I am glad to get feedback, especially from instructors who have class-tested the material. My contact information is on the same page.

The computer output included here is generated automatically (I have automatically edited some lines). You should see what is shown, unless your version differs greatly from mine. This is my *Sage*.

```
1 'Sage Version 6.1.1, Release Date: 2014-02-04'
```

Reading this manual

Here I don't define all the terms or prove all the results. So a student should read the material here after covering the associated chapter in the book, using that for reference.

The association between chapters here and chapters in the book is: *Python and Sage* does not depend on the book, *Gauss's Method* works with Chapter One, *Vector Spaces* is for Chapter Two, *Matrices*, *Maps*, and *Singular Value Decomposition* go with Chapter Three, *Geometry of Linear Maps* goes best with Chapter Four, and *Eigenvalues* fits with Chapter Five (it mentions Jordan Form but only relies on the material up to Diagonalization.)

An instructor may want to make up *Sage* notebooks for the chapters, since the chapters here do not have stand-alone exercises.

[1] See http://www.ams.org/notices/200710/tx071001279p.pdf for the full text. [2] The Free Software Foundation page http://www.gnu.org/philosophy/free-sw.html gives background and a definition. [3] See http://opensource.org/osd.html for a definition.

Acknowledgments

I am glad for this chance to thank the *Sage* Development Team. In particular, without [Sage Development Team, 2012b] this work would not have happened. I am glad also for the chance to mention [Beezer, 2011] as an inspiration. Finally, I am grateful to Saint Michael's College for the time to work on this.

We emphasize practice.
–Shunryu Suzuki [2006]

[A]n orderly presentation is not necessarily bad but by itself may be insufficient.
–Ron Brandt [1998]

Jim Hefferon
Mathematics, Saint Michael's College
Colchester, Vermont USA
2014-Dec-25

Contents

Python and Sage

To work through the Linear Algebra in this manual you must be acquainted with running *Sage*. *Sage* uses the computer language Python so we'll start with that.

Python

Python is a popular computer language,[1] often used for scripting, that is appealing for its simple style and powerful libraries. The significance of 'scripting' is that *Sage* uses it in this way, as a glue to bring together separate parts.

Python is Free. If your operating system doesn't provide it then go to the home page www.python.org and follow the download and installation instructions. Also at that site is Python's excellent tutorial. That tutorial is thorough; here you will see only enough Python to get started. For a more comprehensive introduction see Python Team [2012b].

Comment. There is a new version, Python 3, with some differences. Here we stick to the older version because that is what *Sage* uses. The examples below were produced directly from Python and *Sage* when this manual was generated so they should be exactly what you see,[2] unless your version is quite different than mine. Here is my version of Python.

```
>>> import sys
>>> print sys.version
2.7.6 (default, Mar 22 2014, 22:59:56)
[GCC 4.8.2]
```

Basics Start Python, for instance by entering `python` at a command line. You'll get a couple of lines of information followed by three greater-than characters.

```
>>>
```

This is a prompt. It lets you experiment: if you type Python code and ⟨*Enter*⟩ then the system will read your code, evaluate it, and print the result. We will see below how to write and run whole programs but for now we will experiment. You can always leave the prompt with ⟨*Ctrl*⟩-*D*.

Try entering these expressions (double star is exponentiation).

```
>>> 2 - (-1)
3
>>> 1 + 2*3
7
```

[1] Written by a fan of Monty Python. [2] Lines that are too long are split to fit.

```
>>> 2**3
8
```

Part of Python's appeal is that doing simple things tend to be easy. Here is how you print something to the screen.

```
>>> print 1, "plus", 2, "equals", 3
1 plus 2 equals 3
```

Often you can debug just by putting in commands to print things, and having a straightforward print operator helps with that.

As in any other computer language, variables give you a place to keep values. The first line below puts one in the place called `i` and the second line uses that.

```
>>> i = 1
>>> i + 1
2
```

In some programming languages you must declare the 'type' of a variable before you use it; for instance you would have to declare that i is an integer before you could set $i = 1$. In contrast, Python deduces the type of a variable based on what you do to it — above we assigned 1 to i so Python figured that it must be an integer. Further, we can change how we use the variable and Python will go along; here we change what is in x from an integer to a string.

```
>>> x = 1
>>> x
1
>>> x = 'a'
>>> x
'a'
```

Python lets you assign multiple values simultaneously.

```
>>> first_day, last_day = 0, 365
>>> first_day
0
>>> last_day
365
```

Python computes the right side, left to right, and then assigns those values to the variables on the left. We will often use this construct.

Python complains by *raising an exception* and giving an error message. For instance, we cannot combine a string and an integer.

```
>>> 'a'+1
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
```

The error message's bottom line is the useful one.

Make a comment of the rest of the line with a hash mark `#`.

```
>>> t = 2.2
>>> d = (0.5) * 9.8 * (t**2)  # d in meters
>>> d
23.716000000000005
```

(Comments are more useful in a program than at the prompt.) Programmers often comment an entire line by starting that line with a hash.

As in the listing above, we can get real numbers and even complex numbers.[1]

```
>>> 5.774 * 3
17.322
>>> (3+2j) - (1-4j)
(2+6j)
```

As engineers do, Python uses j for the square root of -1, not i as is traditional in Mathematics.[2]

The examples above show addition, subtraction, multiplication, and exponentiation. Division has an awkward point. Python was originally designed to have the division bar `/` mean real number division when at least one of the two numbers is real. However between two integers the division bar was taken to mean a quotient, as in "2 goes into 5 with quotient 2 and remainder 1."

```
>>> 5.2 / 2.0
2.6
>>> 5.2 / 2
2.6
>>> 5 / 2
2
```

Experience shows this was a mistake. One of the changes in Python 3 is that the quotient operation has become `//` while the single-slash operator is always real division. In the Python 2 we are using, you must make sure that at least one number in a division is real.

```
>>> x = 5
>>> y = 2
>>> (1.0*x) / y
2.5
```

Incidentally, the integer remainder operation (sometimes called 'modulus') uses a percent character: `5 % 2` returns 1.

Variables can also represent truth values; these are *Booleans*.

```
>>> yankees_stink = True
>>> yankees_stink
True
```

You need the initial capital: `True` or `False`, not `true` or `false`.

Above we saw a string consisting of text between single quotes. You can use either single quotes or double quotes, as long as you use the same at both ends of the string. Here `x` and `y` are double-quoted, which makes sense because they contain apostrophes.

```
>>> x = "I'm Popeye the sailor man"
>>> y = "I yam what I yam and that's all what I yam"
>>> x + ', ' + y
"I'm Popeye the sailor man, I yam what I yam and that's all what I yam"
```

[1] Of course these aren't actually real numbers, instead they are floating point numbers, a system that models the reals and is built into your computer's hardware. In the prior example the distinction leaks through since its bottom line ends in 000000000005, marking where the computer's binary approximation does not perfectly match the real that you expect. Similarly Python's integers aren't the integers that you studied in grade school since there is a largest one (give Python the command `import sys` followed by `sys.maxint`). However, while the issue of representation is fascinating — see Python Team [2012a] and Goldberg [1991] — we shall ignore it and just call them integers and reals.

[2] Athough *Sage* lets you use `i`.

The + operation concatenates strings. Inside a double-quoted string use slash-n \n to get a newline.

A string marked by three sets of quotes can contain line breaks.

```
>>> a = """THE ROAD TO WISDOM
...
... The road to wisdom?
... -- Well, it's plain
... and simple to express:
... Err
... and err
... and err again
... but less
... and less
... and less. --Piet Hein"""
```

The three dots at the start of lines after the first is Python's read-eval-print prompt telling you that what you have typed is not complete. We'll see below that a common use for triple-quoted strings is as documentation.

A Python *dictionary* is a finite function. That is, it is a finite set of ordered pairs $\langle key, value \rangle$ subject to the restriction that no key can appear twice. Dictionaries are a simple database.

```
>>> english_words = {'one': 1, 'two': 2, 'three': 3}
>>> english_words['one']
1
>>> english_words['four'] = 4
>>> english_words
{'four': 4, 'three': 3, 'two': 2, 'one': 1}
```

Don't be mislead by this example, the words do not always just come in the reverse of the order in which you entered them. A dictionary's elements will be listed in no apparently-sensible order.

If you assign to an existing key then that will replace the previous value.

```
>>> english_words['one'] = 5
>>> english_words
{'four': 4, 'three': 3, 'two': 2, 'one': 5}
```

Dictionaries are central to Python, in part because looking up values in a dictionary is very fast.

While dictionaries are unordered, a *list* is ordered.

```
>>> a = ['alpha', 'beta', 'gamma']
>>> b = []
>>> c = ['delta']
>>> a
['alpha', 'beta', 'gamma']
>>> a+b+c
['alpha', 'beta', 'gamma', 'delta']
```

Get an element from a list by specifying its index, its place in the list, inside square brackets. Lists are *zero-offset* indexed, that is, the initial element of the list is numbered 0. Count from the back by using negative indices.

```
>>> a[0]
'alpha'
```

```
>>> a[1]
'beta'
>>> a[-1]
'gamma'
```

Specifying two indices separated by a colon gets a *slice* of the list.

```
>>> a[1:3]
['beta', 'gamma']
>>> a[1:2]
['beta']
```

You can add to a list.

```
>>> c.append('epsilon')
>>> c
['delta', 'epsilon']
```

Lists can contain anything, including other lists.

```
>>> x = 4
>>> a = ['alpha', [True, x]]
>>> a
['alpha', [True, 4]]
```

The function `range` returns a list of numbers.

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(1,10)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

By default `range` starts at 0, which is good because lists are zero-indexed. Observe also that 9 is the highest number in the list given by `range(10)`. This makes `range(10)+range(10,20)` give the same list as `range(20)`.

A *tuple* is like a list in that it is ordered.

```
>>> a = ('fee', 'fie', 'foe', 'fum')
>>> a
('fee', 'fie', 'foe', 'fum')
>>> a[0]
'fee'
```

However it is unlike a list in that a tuple is not *mutable*—it cannot change.

```
>>> a[0] = 'phooey'
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

One reason this is useful is that because of it tuples can be keys in dictionaries while list cannot.

```
>>> a = ['ke1az', 5418]
>>> b = ('ke1az', 5418)
>>> d = {a: 'active'}
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
```

```
TypeError: unhashable type: 'list'
>>> d = {b: 'active'}
>>> d
{('ke1az', 5418): 'active'}
```

Python has a special value `None` for when there is no sensible value for a variable. For instance, if your program keeps track of a person's address and includes a variable `apartment` then `None` is the right value for that variable when the person does not live in an apartment.

Flow of control Python supports the traditional ways of affecting the order of statement execution, with a twist.

```
>>> x = 4
>>> if (x == 0):
...       y = 1
... else:
...       y = 0
...
>>> y
0
```

The twist is that while many languages use braces or some other syntax to mark a block of code, Python uses indentation. (Always indent with four spaces.) Here, Python executes the single-line block `y = 1` if x equals 0, otherwise Python sets y to 0.

Notice also that double equals `==` means "is equal to." In contrast, we have already seen that single equals is the assignment operation so that `x = 4` means "x is assigned the value 4."

Python has two variants on the above `if` statement. It could have only one branch

```
>>> x = 4
>>> y = 0
>>> if (x == 0):
...       y = 1
...
>>> y
0
```

or it could have more than two branches.

```
>>> x = 2
>>> if (x == 0):
...       y = 1
... elif (x == 1):
...       y = 0
... else:
...       y = -1
...
>>> y
-1
```

Computers excel at iteration, looping through the same steps.

```
>>> for i in range(5):
...       print i, "squared is", i**2
...
```

```
0 squared is 0
1 squared is 1
2 squared is 4
3 squared is 9
4 squared is 16
```

A `for` loop often involves a `range`.

```
>>> x = [4, 0, 3, 0]
>>> for i in range(len(x)):
...     if (x[i] == 0):
...         print "item",i,"is zero"
...     else:
...         print "item",i,"is nonzero"
...
item 0 is nonzero
item 1 is zero
item 2 is nonzero
item 3 is zero
```

An experienced Python person who was not trying just to illustrate `range` would instead write `for c in x:` since the `for` loop can iterate over any sequence, not just a sequence of integers.

A `for` loop is designed to execute a certain number of times. The natural way to write a loop that will run an uncertain number of times is `while`.

```
>>> n = 27
>>> i = 0
>>> while (n != 1):
...     if (n%2 == 0):
...         n = n / 2
...     else:
...         n = 3*n + 1
...     i = i + 1
...     print "i=", i
...
i= 1
i= 2
i= 3
```

(This listing is incomplete; it takes 111 steps to finish.)[1] Note that "not equal" is `!=`.

The `break` command gets you out of a loop right away.

```
>>> for i in range(10):
...     if (i == 3):
...         break
...     print "i is", i
...
i is 0
i is 1
i is 2
```

A common loop construct is to run through a list performing an action on each entry. Python has a shortcut, *list comprehension*.

[1] The *Collatz conjecture* is that for any starting `n` this loop will terminate; no one knows if it is true.

```
>>> a = [2**i for i in range(4)]
>>> a
[1, 2, 4, 8]
>>> [i-1 for i in a]
[0, 1, 3, 7]
```

Functions A *function* is a group of statements that executes when it is called, and can return values to the caller. Here is a naive version of the quadratic formula.

```
>>> def quad_formula(a, b, c):
...       discriminant = (b**2 - 4*a*c)**(0.5)
...       r1=(-1*b+discriminant) / (2.0*a)
...       r2=(-1*b-discriminant) / (2.0*a)
...       return (r1, r2)
...
>>> quad_formula(1,0,-9)
(3.0, -3.0)
>>> quad_formula(1,2,1)
(-1.0, -1.0)
```

(One way that it is naive is that it doesn't handle complex roots gracefully.)

Functions organize code into blocks. These blocks of code may be run a number of different times, or may belong together conceptually. In a Python program most code is in functions.

At the end of the `def` line, in parentheses, are the function's *parameters*. These can take values passed in by the caller. Functions can have *optional parameters* that have a default value.

```
>>> def hello(name="Jim"):
...       print "Hello,", name
...
>>> hello("Fred")
Hello, Fred
>>> hello()
Hello, Jim
```

Sage uses this aspect of Python heavily.

Functions always return something; if a function never executes a `return` then it will return the value `None`. They can also contain multiple `return` statements, for instance one for an `if` branch and one for an `else`.

Objects and modules In Mathematics, the real numbers is a set associated with some operations such as addition and multiplication. Python is *object-oriented*, which means that we can similarly bundle together data and actions (in this context the functions are called *methods*).

```
>>> class DatabaseRecord(object):
...       def __init__(self, name, age):
...           self.name = name
...           self.age = age
...       def salutation(self):
...           print "Dear", self.name
...
>>> a = DatabaseRecord("Jim", 53)
```

```
>>> a.name
'Jim'
>>> a.age
53
>>> a.salutation()
Dear Jim
>>> b = DatabaseRecord("Fred",109)
>>> b.salutation()
Dear Fred
```

This creates two *instances* of the object `DatabaseRecord`. The `class` code describes what these consist of. The above example uses the *dot notation*: to get the age data for the instance `a` you write `a.age`. (The `self` variable can be puzzling. It is how a method refers to the instance it is part of. Suppose that at the prompt you type `a.name="James"`. Then you've used the name `a` to refer to the instance so you can make the change. In contrast, inside the `class` description code there isn't any fixed instance. The `self` gives you a way to, for example, get the `name` attribute of the current instance.)

You won't be writing your own classes here but you will be using ones from the libraries of code that others have written, including the code for *Sage*, so you must know how to use the classes of others. For instance, Python has a library, or *module*, for math.

```
>>> import math
>>> math.pi
3.141592653589793
>>> math.factorial(5)
120
>>> math.cos(math.pi)
-1.0
```

The `import` statement gets the module and makes its contents available.

Programs The read-eval-print loop is great for small experiments but for more than four or five lines you want to put your work in a separate file and run it as a standalone program.

To write the code, use a text editor (one example is `Emacs`).[1] Use an editor with support for Python such as automatic indentation, and syntax highlighting, where the editor colors your code to make it easier to read.

Here is a first example of a Python program. Start your editor, open a file called `test.py`, and enter these lines. Note the triple-quoted documentation string at the top of the file; include such documentation in everything that you write.

```
# test.py
"""test

A test program for Python.
"""

import datetime

current = datetime.datetime.now()  # get a datetime object
```

[1] It may come with your operating system or see http://www.gnu.org/software/emacs.

```
print "the month number is", current.month
```

Run it through Python (for instance, from the command line run `python test.py`) and you should see output like `the month number is 9`.

Next is a small game. (It uses the Python function `raw_input` that prompts the user and then collects their response.).

```
# guessing_game.py
"""guessing_game

A toy game for demonstration.
"""
import random
CHOICE = random.randint(1,10)

def test_guess(guess):
    """Decide if the guess is correct and print a message.
    """
    if (guess < CHOICE):
        print "  Sorry, your guess is too low"
        return False
    elif (guess > CHOICE):
        print "  Sorry, your guess is too high"
        return False
    print "  You are right!"
    return True

flag = False
while (not flag):
    guess = int(raw_input("Guess an integer between 1 and 10: "))
    flag = test_guess(guess)
```

Here is the output resulting from running this game once at the command line.

```
$ python guessing_game.py
Guess an integer between 1 and 10: 5
  Sorry, your guess is too low
Guess an integer between 1 and 10: 8
  Sorry, your guess is too high
Guess an integer between 1 and 10: 6
  Sorry, your guess is too low
Guess an integer between 1 and 10: 7
  You are right!
```

As earlier, note the triple-quoted documentation strings both for the file as a whole and for the function. They provide information on how to use the `guessing_game.py` program as a whole, and how to use the function inside the program. Go to the directory containing `guessing_game.py` and start the Python read-eval-print loop. At the `>>>` prompt enter `import guessing_game`. You will play through a round of the game (there is a way to avoid this but it doesn't matter here). You are using `guessing_game` as a module. Type `help("guessing_game")`. You will see the documentation, including these lines.

```
DESCRIPTION
```

```
    A toy game for demonstration.

FUNCTIONS
    test_guess(guess)
        Decide if the guess is correct and print a message.
```

Obviously, Python got this from the file's documentation strings. In Python, and also in *Sage*, good practice is to always include documentation that is accessible with the `help` command. All of *Sage*'s built-in routines do this.

Sage

Learning what *Sage* can do is the goal of much of this book so this is only a brief walk-through of preliminaries. See also [Sage Development Team, 2012a] for a more broad-based introduction.

First, if your system does not already supply it then install *Sage* by following the directions at www.sagemath.org.

Command line *Sage*'s command line is like Python's but adapted to mathematical work. First start *Sage*, for instance, enter `sage` into a command line window. You get some initial text and then a prompt (leave the prompt by typing `exit` and ⟨*Enter*⟩.)

```
sage:
```

Experiment with some expressions.

```
sage: 2**3
8
sage: 2^3
8
sage: 3*1 + 4*2
11
sage: 5 == 3+3
False
sage: sin(pi/3)
1/2*sqrt(3)
```

The second expression shows that *Sage* provides a convenient shortcut for exponentiation over Python's `2**3`. The fourth expression shows that *Sage* sometimes returns exact results rather than an approximation. You can still get the approximation; here are three equivalent ways.

```
sage: sin(pi/3).numerical_approx()
0.866025403784439
sage: sin(pi/3).n()
0.866025403784439
sage: n(sin(pi/3))
0.866025403784439
```

The function `n()` is an abbreviation for `numerical_approx()`.

Script You can group *Sage* commands together in a file. This way you can test the commands, and also reuse them without having to retype.

Create a file with the extension `.sage`, such as `sage_normal.sage`. Enter this function and save the file.

```
def normal_curve(upper_limit):
    """Approximate area under the Normal curve from 0 to upper_limit.
    """
    stdev = 1.0
    mu = 0.0
    area=numerical_integral((1/sqrt(2*pi) * e^(-0.5*(x)^2)),
                            0, upper_limit)
    print "area is", area[0]
```

Bring in the file with a `runfile` command and then you can use the commands defined there.

```
sage: runfile("sage_normal.sage")
sage: normal_curve(1.0)
```

Notebook *Sage* also offers a browser-based interface, where you can set up worksheets to run alone or with other people, where you can easily view plots integrated with the text, and many other nice features.

From the *Sage* prompt run `notebook()` and work through the tutorial.

Gauss's Method

Sage can solve linear systems in a number of ways. The first way is to use a general system solver, one not specialized to linear systems. The second way is to leverage the special advantages of linear systems. We'll see both.

Systems

To enter a system of equations you must first enter single equations. So you must start with variables. We have seen one kind of variable in giving commands like these.

```
sage: x = 3
sage: 7*x
21
```

Here x is the name of a location to hold values. Variables in equations are something different; in the equation $C = 2\pi \cdot r$ the two variables do not have fixed values, nor are they tied to a location in the computer's memory.

To illustrate the difference enter an unassigned variable.[1]

```
sage: y
---------------------------------------------------------------------------
                                                                        ----
NameError                                 Traceback (most recent call
                                                                       last)
<ipython-input-1-009520053b00> in <module>()
----> 1 y

NameError: name 'y' is not defined
```

Sage defaults to expecting that y is the name of a location to hold a value. Before you use it as a symbolic variable, you must first warn the system.

```
sage: var('x,y')
(x, y)
```

[1] These output blocks are taken automatically from *Sage*'s responses, without any by-hand copying and pasting, ensuring that this manual shows what *Sage* actually does. But this has two drawbacks. First, some of *Sage*'s response lines are too long to fit in the block. Sometimes these lines will be split and have their tail carried to the next line, as here, while sometimes the response lines get truncated if the extra characters are a distraction. The second drawback is that *Sage* occasionally gives system-specific responses, say by naming the directory of a file with an error. Naturally on your system it will show your directory. This is the first time in this manual that the issue of these artifacts appears; we won't note them again.

```
sage: y
y
sage: 2*y
2*y
sage: k = 3
sage: 2*k
6
```

Because we haven't told *Sage* otherwise, it takes `k` as a location to hold values and it evaluates `2*k`. But it does not evaluate `2*y` because `y` is a symbolic variable.

With that, a system of equations is a list.

```
sage: var('x,y,z')
(x, y, z)
sage: eqns = [x-y+2*z == 4, 2*x+2*y == 12, x-4*z==5]
```

You must write double equals `==` in the equations instead of the assignment operator `=`. Also, you must write `2*x` instead of `2x`. Either mistake will trigger a `SyntaxError: invalid syntax`.

Solve the system with *Sage*'s general-purpose solver.

```
sage: eqns = [x-y+2*z == 4, 2*x+2*y == 12, x-4*z==5]
sage: solve(eqns, x,y,z)
[[x == 5, y == 1, z == 0]]
```

You can put a parameter in the right side and solve for the variables in terms of the parameter.

```
sage: var('x,y,z,a')
(x, y, z, a)
sage: eqns = [x-y+2*z == a, 2*x+2*y == 12, x-4*z==5]
sage: solve(eqns, x,y,z)
[[x == 2/5*a + 17/5, y == -2/5*a + 13/5, z == 1/10*a - 2/5]]
```

Matrices The *solve* routine is general-purpose but for the special case of linear systems matrix notation is the best tool.

Most of the matrices in the book have entries that are fractions so here we stick to those. *Sage* uses 'QQ' for the rational numbers, 'RR' or 'RDF' for real numbers (the first are arbitrary precision reals while the second are double-length floats; for most practical calculations the second type is best), 'CC' or 'CDF' for the complex numbers (arbitrary precision or double floats), and 'ZZ' for the integers.

Enter a row of the matrix as a list, and enter the entire matrix as a list of rows.

```
sage: M = matrix(QQ, [[1, 2, 3], [4, 5, 6], [7 ,8, 9]])
sage: M
[1 2 3]
[4 5 6]
[7 8 9]
sage: M[1,2]
6
sage: M.nrows()
3
sage: M.ncols()
3
```

Sage lists are zero-indexed, as with Python lists, so `M[1,2]` asks for the entry in the second row and third column.

Enter a vector in much the same way.

```
sage: v = vector(QQ, [2/3, -1/3, 1/2])
sage: v
(2/3, -1/3, 1/2)
sage: v[1]
-1/3
```

Sage does not worry too much about the distinction between row and column vectors. Note however that it appears with rounded brackets so that it looks different than a one-row matrix.

You can augment a matrix with a vector.

```
sage: M = matrix(QQ, [[1, 2, 3], [4, 5, 6], [7 ,8, 9]])
sage: v = vector(QQ, [2/3, -1/3, 1/2])
sage: M_prime = M.augment(v)
sage: M_prime
[   1    2    3  2/3]
[   4    5    6 -1/3]
[   7    8    9  1/2]
```

You can use an optional argument to have *Sage* remember the distinction between the two parts of M'.

```
sage: M_prime = M.augment(v,subdivide=True)
sage: M_prime
[   1    2    3| 2/3]
[   4    5    6|-1/3]
[   7    8    9| 1/2]
```

Row operations Computers are good for jobs that are tedious and error-prone. Row operations are both.

```
sage: M = matrix(QQ, [[0, 2, 1], [2, 0, 4], [2 ,-1/2, 3]])
sage: v = vector(QQ, [2, 1, -1/2])
sage: M_prime = M.augment(v, subdivide=True)
sage: M_prime
[   0    2    1|   2]
[   2    0    4|   1]
[   2 -1/2    3|-1/2]
```

Swap the top rows (remember that row indices start at zero).

```
sage: M_prime.swap_rows(0,1)
sage: M_prime
[   2    0    4|   1]
[   0    2    1|   2]
[   2 -1/2    3|-1/2]
```

Rescale the top row.

```
sage: M_prime.rescale_row(0, 1/2)
sage: M_prime
```

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

Get a new bottom row by adding -2 times the top to the current bottom row.

```
sage: M_prime.add_multiple_of_row(2,0,-2)
sage: M_prime
[    1     0     2| 1/2]
[    0     2     1|   2]
[    0  -1/2    -1|-3/2]
```

Finish by finding echelon form.

```
sage: M_prime.add_multiple_of_row(2,1,1/4)
sage: M_prime
[    1     0     2| 1/2]
[    0     2     1|   2]
[    0     0  -3/4|  -1]
```

By the way, *Sage* would have given us echelon form in a single operation had we run the command `M_prime.echelon_form()`.

Now by-hand back substitution gives the solution, or we can use `solve`.

```
sage: var('x,y,z')
(x, y, z)
sage: eqns=[-3/4*z == -1, 2*y+z == 2, x+2*z == 1/2]
sage: solve(eqns, x, y, z)
[[x == (-13/6), y == (1/3), z == (4/3)]]
```

The operations `swap_rows`, `rescale_rows`, and `add_multiple_of_row` changed the matrix M'. *Sage* has related commands that return a changed matrix but leave the starting matrix unchanged.

```
sage: M = matrix(QQ, [[1/2, 1, -1], [1, -2, 0], [2 ,-1, 1]])
sage: v = vector(QQ, [0, 1, -2])
sage: M_prime = M.augment(v, subdivide=True)
sage: M_prime
[1/2   1  -1|  0]
[  1  -2   0|  1]
[  2  -1   1| -2]
sage: N = M_prime.with_rescaled_row(0,2)
sage: M_prime
[1/2   1  -1|  0]
[  1  -2   0|  1]
[  2  -1   1| -2]
sage: N
[ 1  2 -2| 0]
[ 1 -2  0| 1]
[ 2 -1  1|-2]
```

Here, M' is unchanged by the routine while N is the returned changed matrix. The other two routines of this kind are `with_swapped_rows` and `with_added_multiple_of_row`.

Nonsingular and singular systems Resorting to `solve` after going through the row operations is artless. *Sage* will give reduced row echelon form straight from the augmented matrix.

```
sage: M = matrix(QQ, [[1/2, 1, -1], [1, -2, 0], [2 ,-1, 1]])
sage: v = vector(QQ, [0, 1, -2])
sage: M_prime = M.augment(v, subdivide=True)
sage: M_prime.rref()
[      1       0       0|   -4/5]
[      0       1       0|  -9/10]
[      0       0       1|-13/10]
```

In that example the matrix M on the left is nonsingular because it is square and because Gauss's Method produces echelon forms where every one of M's columns has a leading variable. The next example starts with a different square matrix, a singular matrix, and consequently leads to echelon form systems where there are columns on the left that do not have a leading variable.

```
sage: M = matrix(QQ, [[1, 1, 1], [1, 2, 3], [2 , 3, 4]])
sage: v = vector(QQ, [0, 1, 1])
sage: M_prime = M.augment(v, subdivide=True)
sage: M_prime
[1 1 1|0]
[1 2 3|1]
[2 3 4|1]
sage: M_prime.rref()
[ 1  0 -1|-1]
[ 0  1  2| 1]
[ 0  0  0| 0]
```

Recall that the singular case has two subcases. The first is above: because in echelon form every row that is all zeros on the left has an entry on the right that is also zero, the system has infinitely many solutions. In contrast, with the same starting matrix the example below has a row that is zeros on the left but is nonzero on the right and so the system has no solution.

```
sage: v = vector(QQ, [0, 1, 2])
sage: M_prime = M.augment(v, subdivide=True)
sage: M_prime.rref()
[ 1  0 -1| 0]
[ 0  1  2| 0]
[ 0  0  0| 1]
```

The difference between the subcases has to do with the relationships among the rows of M and the relationships among the rows of the vector. In both cases, the relationship among the rows of the matrix M is that the first two rows add to the third. In the first case the vector has the same relationship while in the second it does not.

The easy way to ensure that a zero row in the matrix on the left is associated with a zero entry in the vector on the right is to make the vector have all zeros, that is, to consider the homogeneous system associated with M.

```
sage: v = zero_vector(QQ, 3)
sage: v
(0, 0, 0)
sage: M = matrix(QQ, [[1, 1, 1], [1, 2, 3], [2 , 3, 4]])
sage: M_prime = M.augment(v, subdivide=True)
```

```
sage: M_prime
[1  1  1|0]
[1  2  3|0]
[2  3  4|0]
sage: M_prime.rref()
[ 1  0 -1| 0]
[ 0  1  2| 0]
[ 0  0  0| 0]
```

You can get the numbers of the columns having leading entries with the `pivots` method (there is a complementary `nonpivots`).

```
sage: M_prime
[1  1  1|0]
[1  2  3|1]
[2  3  4|2]
sage: M_prime.pivots()
(0, 1, 3)
sage: M_prime.rref()
[ 1  0 -1| 0]
[ 0  1  2| 0]
[ 0  0  0| 1]
```

Because column 2 is not in the list of pivots we know that the system is singular before we see it in echelon form.

We can use this observation to write a routine that decides if a square matrix is nonsingular.

```
sage: def check_nonsingular(mat):
....:     if not(mat.is_square()):
....:         print "ERROR: mat must be square"
....:         return
....:     p = mat.pivots()
....:     for col in range(mat.ncols()):
....:         if not(col in p):
....:             print "nonsingular"
....:             break
....:
sage: N = Matrix(QQ, [[1, 2, 3], [4, 5, 6], [7, 8, 9]])
sage: check_nonsingular(N)
nonsingular
sage: N = Matrix(QQ, [[1, 0, 0], [0, 1, 0], [0, 0, 1]])
sage: check_nonsingular(N)
```

Actually, *Sage* matrices already have a method `is_singular` but this illustrates how you can write routines to extend *Sage*.

Parametrization You can use `solve` to give the solution set of a system with infinitely many solutions. Start with the square matrix of coefficients from above, where the top two rows add to the bottom row, and adjoin a vector satisfying the same relationship to get a system with infinitely many solutions. Then convert that to a system of equations.

```
sage: M = matrix(QQ, [[1, 1, 1], [1, 2, 3], [2 , 3, 4]])
sage: v = vector(QQ, [1, 0, 1])
```

```
sage: M_prime = M.augment(v, subdivide=True)
sage: M_prime
[1 1 1|1]
[1 2 3|0]
[2 3 4|1]
sage: var('x,y,z')
(x, y, z)
sage: eqns = [x+y+z == 1, x+2*y+3*z == 0, 2*x+3*y+4*z == 1]
sage: solve(eqns, x, y)
[[x == z + 2, y == -2*z - 1]]
sage: solve(eqns, x, y, z)
[[x == r1 + 2, y == -2*r1 - 1, z == r1]]
```

The first of the two `solve` calls asks *Sage* to solve only for `x` and `y` and so the solution is in terms of z. In the second call *Sage* produces a parameter of its own.

Automation

We finish by showing two routines to automate the by-hand row reductions of the kind that the text has in the homework. These use the matrix capabilities of *Sage* to both describe and perform the row operations that bring a matrix to echelon form or to reduced echelon form.

Loading and running The source file of the script is below, at the end. First here are a few sample calls. Start *Sage* in the directory containing the file `gauss_method.sage`.

```
sage: runfile("gauss_method.sage")
sage: M = matrix(QQ, [[1/2, 1, 4], [2, 4, -1], [1, 2, 0]])
sage: v = vector(QQ, [-2, 5, 4])
sage: M_prime = M.augment(v, subdivide=True)
sage: gauss_method(M_prime)
[1/2   1   4| -2]
[  2   4  -1|  5]
[  1   2   0|  4]
take -4 times row 1 plus row 2
take -2 times row 1 plus row 3
[1/2   1   4| -2]
[  0   0 -17| 13]
[  0   0  -8|  8]
take -8/17 times row 2 plus row 3
[  1/2     1     4|   -2]
[    0     0   -17|   13]
[    0     0     0|32/17]
```

Because the matrix has rational number elements the operations are exact — without floating point issues.

The remaining examples skip the steps to make an augmented matrix.

```
sage: M1 = matrix(QQ, [[2, 0, 1, 3], [-1, 1/2, 3, 1], [0, 1, 7, 5]])
sage: gauss_method(M1)
[  2   0   1   3]
```

```
[ -1 1/2   3   1]
[  0   1   7   5]
take 1/2 times row 1 plus row 2
[  2   0   1   3]
[  0 1/2 7/2 5/2]
[  0   1   7   5]
take -2 times row 2 plus row 3
[  2   0   1   3]
[  0 1/2 7/2 5/2]
[  0   0   0   0]
```

The script also has a routine to go all the way to reduced echelon form.

```
sage: r1 = [1, 2, 3, 4]
sage: r2 = [1, 2, 3, 4]
sage: r3 = [2, 4, -1, 5]
sage: r4 = [1, 2, 0, 4]
sage: M2 = matrix(QQ, [r1, r2, r3, r4])
sage: gauss_jordan(M2)
[ 1  2  3  4]
[ 1  2  3  4]
[ 2  4 -1  5]
[ 1  2  0  4]
take -1 times row 1 plus row 2
take -2 times row 1 plus row 3
take -1 times row 1 plus row 4
[ 1  2  3  4]
[ 0  0  0  0]
[ 0  0 -7 -3]
[ 0  0 -3  0]
swap row 2 with row 3
[ 1  2  3  4]
[ 0  0 -7 -3]
[ 0  0  0  0]
[ 0  0 -3  0]
take -3/7 times row 2 plus row 4
[  1   2   3   4]
[  0   0  -7  -3]
[  0   0   0   0]
[  0   0   0 9/7]
swap row 3 with row 4
[  1   2   3   4]
[  0   0  -7  -3]
[  0   0   0 9/7]
[  0   0   0   0]
take -1/7 times row 2
take 7/9 times row 3
[  1   2   3   4]
[  0   0   1 3/7]
[  0   0   0   1]
[  0   0   0   0]
take -4 times row 3 plus row 1
```

```
take -3/7 times row 3 plus row 2
[1 2 3 0]
[0 0 1 0]
[0 0 0 1]
[0 0 0 0]
take -3 times row 2 plus row 1
[1 2 0 0]
[0 0 1 0]
[0 0 0 1]
[0 0 0 0]
```

These are naive implementations of Gauss's Method that are just for fun. (For instance they don't handle real numbers, just rationals.) But they do illustrate the point made in this manual's Preface, since a person builds intuition by doing a reasonable number of reasonably hard Gauss's Method reductions by hand, and at that point automation can take over.

Source of `gauss_method.sage` *Code comment:* lines 50 and 88 are too long for this page so they end with a slash \ to make Python continue on the next line.

```
# code gauss_method.sage
# Show Gauss's method and Gauss-Jordan reduction steps.
# 2012-Apr-20  Jim Hefferon  Public Domain.

# Naive Gaussian reduction
def gauss_method(M,rescale_leading_entry=False):
    """Describe the reduction to echelon form of the given matrix of
    rationals.

    M  matrix of rationals   e.g., M = matrix(QQ, [[..], [..], ..])
    rescale_leading_entry=False  boolean  make leading entries to 1's

    Returns: None.  Side effect: M is reduced.  Note that this is
    echelon form, not reduced echelon form;, and that this routine
    does not end the same way as does M.echelon_form().

    """
    num_rows=M.nrows()
    num_cols=M.ncols()
    print M

    col = 0   # all cols before this are already done
    for row in range(0,num_rows):
        # ?Need to swap in a nonzero entry from below
        while (col < num_cols
               and M[row][col] == 0):
      for i in M.nonzero_positions_in_column(col):
          if i > row:
                    print " swap row",row+1,"with row",i+1
        M.swap_rows(row,i)
                    print M
                    break
            else:
```

```
                col += 1

  if col >= num_cols:
            break

        # Now guaranteed M[row][col] != 0
        if (rescale_leading_entry
           and M[row][col] != 1):
            print " take",1/M[row][col],"times row",row+1
            M.rescale_row(row,1/M[row][col])
      print M
        change_flag=False
        for changed_row in range(row+1,num_rows):
            if M[changed_row][col] != 0:
                change_flag=True
                factor=-1*M[changed_row][col]/M[row][col]
                print " take",factor,"times row",row+1,  \
                      "plus row",changed_row+1
                M.add_multiple_of_row(changed_row,row,factor)
        if change_flag:
            print M
        col +=1

# Naive Gauss-Jordan reduction
def gauss_jordan(M):
    """Describe the reduction to reduced echelon form of the
    given matrix of rationals.

    M  matrix of rationals   e.g., M = matrix(QQ, [[..], [..], ..])

    Returns: None.  Side effect: M is reduced.

    """
    gauss_method(M,rescale_leading_entry=False)
    # Get list of leading entries [le in row 0, le in row1, ..]
    pivot_list=M.pivots()
    # Rescale leading entries
    change_flag=False
    for row in range(0,len(pivot_list)):
        col=pivot_list[row]
        if M[row][col] != 1:
            change_flag=True
            print " take",1/M[row][col],"times row",row+1
            M.rescale_row(row,1/M[row][col])
    if change_flag:
        print M
    # Pivot
    for row in range(len(pivot_list)-1,-1,-1):
        col=pivot_list[row]
        change_flag=False
        for changed_row in range(0,row):
```

```
85              if M[changed_row,col] != 0:
86                  change_flag=True
87                  factor=-1*M[changed_row][col]/M[row][col]
88                  print "  take",factor,"times  row",    \
89                        row+1,"plus  row",changed_row+1
90                  M.add_multiple_of_row(changed_row,row,factor)
91          if  change_flag:
92              print M
```

Vector Spaces

Sage can operate with vector spaces, for example by finding a basis for a space.

In this chapter vector spaces take scalars from the reals numbers $\mathbb{R}$ (the book's final chapter uses the complex numbers). You can choose from two models of the real numbers. One is `RDF`, the computer's built-in floating point model of real numbers.[1] The other is `RR`, an arbitrary precision model of reals.[2] The second model is useful in some circumstances but the first runs faster and is more widely used in practice so that is what we will use here.

Real n-spaces

Sage allows us to create vector spaces. Here is $\mathbb{R}^3$.

```
sage: V = VectorSpace(RDF,3)
sage: V
Vector space of dimension 3 over Real Double Field
```

Next we can try subspaces. The plane through the origin in $\mathbb{R}^3$ described by the equation $x-2y+2z=0$ is a subspace of $\mathbb{R}^3$. You can describe it as a span.

$$W=\{\begin{pmatrix}x\\y\\z\end{pmatrix}\mid x=2y-2z\}=\{\begin{pmatrix}2\\1\\0\end{pmatrix}y+\begin{pmatrix}-2\\0\\1\end{pmatrix}z\mid y,z\in\mathbb{R}\}$$

In addition to creating that subspace, you can test membership.

```
sage: V = VectorSpace(RDF,3)
sage: v1 = vector(RDF, [2, 1, 0])
sage: v2 = vector(RDF, [-2, 0, 1])
sage: W = V.span([v1, v2])
sage: v3 = vector(RDF, [0, 1, 1])
sage: v3 in W
True
sage: v4 = vector(RDF, [1, 0, 0])
sage: v4 in W
False
```

For a closer look at what's happening, look at this subspace of $\mathbb{R}^4$.

[1] This computer uses IEEE 754 double-precision binary floating-point numbers; if you have programmed then you may know this number model as binary64. [2] It defaults to 53 bits of precision to reproduce double-precision computations.

```
sage: V = VectorSpace(RDF,4)
sage: V
Vector space of dimension 4 over Real Double Field
sage: v1 = vector(RR, [2, 0, -1, 0])
sage: W = V.span([v1])
sage: W
Vector space of degree 4 and dimension 1 over Real Double Field
Basis matrix:
[ 1.0  0.0 -0.5  0.0]
```

As part of creating it, *Sage* has identified the dimension of the subspace and found a basis containing one vector (it prefers the vector with a leading 1).

As earlier, the membership set relation works here.

```
sage: v2 = vector(RDF, [2, 1, -1, 0])
sage: v2 in W
False
sage: v3 = vector(RDF, [-4, 0, 2, 0])
sage: v3 in W
True
```

Basis *Sage* has a command to retrieve a basis for a space.

```
sage: V = VectorSpace(RDF,2)
sage: v = vector(RDF, [1, -1])
sage: W = V.span([v])
sage: W.basis()
[
(1.0, -1.0)
]
```

This is the basis you will see if you ask for a description of W.

```
sage: W
Vector space of degree 2 and dimension 1 over Real Double Field
Basis matrix:
[ 1.0 -1.0]
```

Of course, there are subspaces with bases of size greater than one.

```
sage: V = VectorSpace(RDF,3)
sage: v1 = vector(RDF, [1, -1, 0])
sage: v2 = vector(RDF, [1, 1, 0])
sage: W = V.span([v1, v2])
sage: W.basis()
[
(1.0, 0.0, 0.0),
(0.0, 1.0, 0.0)
]
```

Adding a linearly dependent vector $\vec{v}_3$ to the spanning set doesn't change the space.

```
sage: W = V.span([v1, v2])
sage: v3 = vector(RDF, [2, 3, 0])
```

```
sage: W_prime = V.span([v1, v2, v3])
sage: W_prime.basis()
[
(1.0, 0.0, 0.0),
(0.0, 1.0, 0.0)
]
```

In the prior example, notice that *Sage* does not simply give you back the linearly independent vectors that you gave it. Instead, by default *Sage* takes the vectors from the spanning set as the rows of a matrix, brings that matrix to reduced echelon form, and reports the nonzero rows as the members of the basis. Because each matrix has one and only one reduced echelon form, each vector subspace of real n-space has one and only one such basis; this is the *canonical basis* for the space.

It is this basis that *Sage* shows when you ask for a description of the space.

```
sage: W = V.span([v1, v2])
sage: W
Vector space of degree 3 and dimension 2 over Real Double Field
Basis matrix:
[1.0 0.0 0.0]
[0.0 1.0 0.0]
sage: v3 = vector(RDF, [2, 3, 0])
sage: W_prime = V.span([v1, v2, v3])
sage: W_prime
Vector space of degree 3 and dimension 2 over Real Double Field
Basis matrix:
[1.0 0.0 0.0]
[0.0 1.0 0.0]
```

If you are keen on your own basis then *Sage* will accommodate you.

```
sage: V = VectorSpace(RDF,3)
sage: v1 = vector(RDF, [1, 2, 3])
sage: v2 = vector(RDF, [2, 1, 3])
sage: W = V.span_of_basis([v1, v2])
sage: W.basis()
[
(1.0, 2.0, 3.0),
(2.0, 1.0, 3.0)
]
sage: W
Vector space of degree 3 and dimension 2 over Real Double Field
User basis matrix:
[1.0 2.0 3.0]
[2.0 1.0 3.0]
```

Equality You can test whether spaces are equal.

```
sage: V = VectorSpace(RDF,4)
sage: v1 = vector(RDF, [1, 0, 0, 0])
sage: v2 = vector(RDF, [1, 1, 0, 0])
sage: W12 = V.span([v1, v2])
```

```
sage: v3 = vector(RDF, [2, 1, 0, 0])
sage: W13 = V.span([v1, v3])
```

One way to assure yourself that the two spaces $W_{1,2}$ and $W_{1,3}$ are equal is to use set membership.

```
sage: v3 in W12
True
sage: v2 in W13
True
```

Then, since obviously $\vec{v}_1 \in W_{1,2}$ and $\vec{v}_1 \in W_{1,3}$, the two spans are equal.

But the more straightforward way to test equality is to just ask *Sage*.

```
sage: W12 == W13
True
```

The exercise of == equality would be half-hearted without a test of inequality.

```
sage: v4 = vector(RDF, [1, 1, 1, 1])
sage: W14 = V.span([v1, v4])
sage: v2 in W14
False
sage: v3 in W14
False
sage: v4 in W12
False
sage: v4 in W13
False
sage: W12 == W14
False
sage: W13 == W14
False
```

This illustrates a point about algorithms. *Sage* could check for equality of two spans by checking whether every member of the first spanning set is in the second space and vice versa (since the two spanning sets are finite). But *Sage* does something different. For each space it maintains the canonical basis and it checks for equality of spaces just by checking whether they have the same canonical bases. These two algorithms have the same external behavior, in that both decide whether two spaces are equal. But the algorithms differ internally, and as a result the second is faster. Finding the fastest way to do jobs is an important research area of computing.

Operations *Sage* finds the intersection of two spaces. Consider these members of $\mathbb{R}^3$.

$$\vec{v}_1 = \begin{pmatrix}1\\0\\0\end{pmatrix} \quad \vec{v}_2 = \begin{pmatrix}0\\1\\0\end{pmatrix} \quad \vec{v}_3 = \begin{pmatrix}0\\0\\3\end{pmatrix}$$

Form two spans, the xy-plane $W_{1,2} = [\vec{v}_1, \vec{v}_2]$ and the yz-plane $W_{2,3} = [\vec{v}_2, \vec{v}_3]$. The intersection of these two is the y-axis.

```
sage: V = VectorSpace(RDF,3)
sage: v1 = vector(RDF, [1, 0, 0])
sage: v2 =  vector(RDF, [0, 1, 0])
sage: W12 = V.span([v1, v2])
```

```
sage: W12.basis()
[
(1.0, 0.0, 0.0),
(0.0, 1.0, 0.0)
]
sage: v3 = vector(RDF, [0, 0, 2])
sage: W23 = V.span([v2, v3])
sage: W23.basis()
[
(0.0, 1.0, 0.0),
(0.0, 0.0, 1.0)
]
sage: W = W12.intersection(W23)
sage: W.basis()
[
(0.0, 1.0, 0.0)
]
```

Remember that the trivial space $\{\vec{0}\}$ is the span of the empty set.

```
sage: v3 = vector(RDF, [1, 1, 1])
sage: W3 = V.span([v3])
sage: W3.basis()
[
(1.0, 1.0, 1.0)
]
sage: W4 = W12.intersection(W3)
sage: W4.basis()
[

]
```

Sage will also find the sum of spaces, the span of their union.

```
sage: W5 = W12 + W3
sage: W5.basis()
[
(1.0, 0.0, 0.0),
(0.0, 1.0, 0.0),
(0.0, 0.0, 1.0)
]
sage: W5 == V
True
sage: W5
Vector space of degree 3 and dimension 3 over Real Double Field
Basis matrix:
[1.0 0.0 0.0]
[0.0 1.0 0.0]
[0.0 0.0 1.0]
```

Other spaces

These computations extend to vector spaces that aren't a subspace of some $\mathbb{R}^n$ by finding a real space that is just like the one you have.

Consider this vector space of quadratic polynomials, under the usual operations of polynomial addition and scalar multiplication.

$$\{a_2x^2+a_1x+a_0 \mid a_2=a_0+a_1\}=\{(a_1+a_0)x^2+a_1x+a_0 \mid a_1,a_0\in\mathbb{R}\}$$

It is just like this subspace of $\mathbb{R}^3$.[1]

$$\{\begin{pmatrix}a_1+a_0\\a_1\\a_0\end{pmatrix} \mid a_1,a_0\in\mathbb{R}\}=\{\begin{pmatrix}1\\1\\0\end{pmatrix}a_1+\begin{pmatrix}1\\0\\1\end{pmatrix}a_0 \mid a_1,a_0\in\mathbb{R}\}$$

```
sage: V = VectorSpace(RDF,3)
sage: v1 = vector(RDF, [1, 1, 0])
sage: v2 = vector(RDF, [1, 0, 1])
sage: W = V.span([v1, v2])
sage: W.basis()
[
(1.0, 0.0, 1.0),
(0.0, 1.0, -1.0)
]
```

Similarly you can represent this space of 2×2 matrices

$$\{\begin{pmatrix}a&b\\c&d\end{pmatrix} \mid a-b+c=0 \text{ and } b+d=0\}$$

by finding a real 4-space just like it. Rewrite the given space

$$\{\begin{pmatrix}a&b\\c&d\end{pmatrix} \mid a=-c-d \text{ and } b=-d\}=\{\begin{pmatrix}-1&0\\1&0\end{pmatrix}c+\begin{pmatrix}-1&-1\\0&1\end{pmatrix}d \mid c,d\in\mathbb{R}\}$$

and then here is a natural matching real space.

```
sage: V = VectorSpace(RDF,4)
sage: v1 = vector(RDF, [-1, 0, 1, 0])
sage: v2 = vector(RDF, [-1, -1, 0, 1])
sage: W = V.span([v1, v2])
sage: W.basis()
[
(1.0, -0.0, -1.0, -0.0),
(0.0, 1.0, 1.0, -1.0)
]
```

You could have gotten another matching space by going down the columns instead of across the rows.

[1] The textbook's chapter on Maps Between Spaces makes "just like" precise.

```
sage: V = VectorSpace(RDF,4)
sage: v1 = vector(RDF, [-1, 1, 0, 0])
sage: v2 = vector(RDF, [-1, 0, -1, 1])
sage: W = V.span([v1, v2])
sage: W.basis()
[
(1.0, 0.0, 1.0, -1.0),
(0.0, 1.0, 1.0, -1.0)
]
```

There are still other ways to produce a matching space. They look different than each other but the important things about the spaces, such as dimension, are unaffected by their look. That is the subject of the book's third chapter.

Matrices

Matrix operations are mechanical, and are therefore perfectly suited for mechanizing.

Defining

To define a matrix you can use real number entries, or complex entries, or entries from other number systems such as the rationals.

```
sage: A = matrix(RDF, [[1, 2], [3, 4]])
sage: A
[1.0 2.0]
[3.0 4.0]
sage: i = CDF(i)
sage: A = matrix(CDF, [[1+2*i, 3+4*i], [5+6*i, 7+8*i]])
sage: A
[1.0 + 2.0*I 3.0 + 4.0*I]
[5.0 + 6.0*I 7.0 + 8.0*I]
sage: A = matrix(QQ, [[1, 2], [3, 4]])
sage: A
[1 2]
[3 4]
```

Here we represent reals with the model connected to the computer's hardware floating point implementation. For the complex numbers we've again used floating points but there is an additional consideration. By default *Sage* uses `i` for the square root of -1 (in contrast with Python, which uses `j`). Note that before working with complex numbers we reset `i` because that letter is used for many things — perhaps earlier in a session we used it for the index of an array and we want it to lose that prior value — so resetting is a good practice.

Unless we have a reason to do otherwise, in this chapter we'll use rational numbers because the matrices are easier to read — 1 is easier than 1.0 — and because the matrices in the book usually have rational entries.

The `matrix` constructor allows you to specify the number of rows and columns.

```
sage: B = matrix(QQ, 2, 3, [[1, 1, 1], [2, 2, 2]])
sage: B
[1 1 1]
[2 2 2]
```

If your specified size doesn't match the entries you list

```
sage: B = matrix(QQ, 3, 3, [[1, 1, 1], [2, 2, 2]])
```

then *Sage*'s error says `Number of rows does not match up with specified number`. Until now we've let *Sage* figure out the matrix's number of rows and columns size from the entries but a shortcut to get the zero matrix is to put the number zero in the place of the entries, and there you must say which size you want.

```
sage: B = matrix(QQ, 2, 3, 0)
sage: B
[0 0 0]
[0 0 0]
```

Another place where specifying the size is a convenience is *Sage*'s shortcut to get an identity matrix.

```
sage: B = matrix(QQ, 2, 2, 1)
sage: B
[1 0]
[0 1]
```

The difference between this shortcut and the prior one is that `matrix(QQ, 3, 2, 1)` gives an error because an identity matrix must be square. *Sage* has another shortcut that can't lead to this error.

```
sage: I = identity_matrix(4)
sage: I
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
```

Sage has a wealth of methods on matrices. For instance, you can transpose the rows to columns or test if the matrix is *symmetric*, unchanged by transposition.

```
sage: A = matrix(QQ, [[1, 2], [3, 4]])
sage: A.transpose()
[1 3]
[2 4]
sage: A.is_symmetric()
False
```

Linear combinations

Addition and subtraction are natural.

```
sage: A = matrix(QQ, [[1, 2], [3, 4]])
sage: B = matrix(QQ, [[1, 1], [2, -2]])
sage: A+B
[2 3]
[5 2]
sage: A-B
[0 1]
```

```
[1 6]
sage: B-A
[ 0 -1]
[-1 -6]
```

Sage knows that adding matrices with different sizes is undefined; this gives an error.

```
sage: A = matrix(QQ, [[1, 2], [3, 4]])
sage: C = matrix(QQ, [[0, 0, 2], [3, 2, 1]])
sage: A+C
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call
<ipython-input-3-4f8d12176c8f> in <module>()
----> 1 A+C

/usr/lib/sagemath/local/lib/python2.7/site-packages/sage/structure/ele

/usr/lib/sagemath/local/lib/python2.7/site-packages/sage/structure/coe

TypeError: unsupported operand parent(s) for '+': 'Full MatrixSpace of
               2 by 2 dense matrices over Rational Field' and 'Full
               MatrixSpace of 2 by 3 dense matrices over Rational Field'
```

Some of the lines in that output block don't contain useful information so they've just been truncated. But the last line, which is so long it had to be broken and wrapped twice, is where the action is. It says that the `+` operand is not defined between a 2×2 matrix and a 2×3 matrix.

Scalar multiplication is also natural, so you have linear combinations.

```
sage: 3*A
[ 3  6]
[ 9 12]
sage: 3*A-4*B
[-1  2]
[ 1 20]
```

Multiplication

Matrix-vector product Matrix-vector multiplication is just what you would guess.

```
sage: A = matrix(QQ, [[1, 3, 5, 9], [0, 2, 4, 6]])
sage: v = vector(QQ, [1, 2, 3, 4])
sage: A*v
(58, 40)
```

The 2×4 matrix A multiplies the 4×1 column vector $\vec{v}$, with the vector on the right side, as $A\vec{v}$.

If you try this vector on the left as `v*A` then *Sage* gives a mismatched-sizes error.

```
sage: A = matrix(QQ, [[1, 3, 5, 9], [0, 2, 4, 6]])
sage: v = vector(QQ, [1, 2, 3, 4])
sage: v*A
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call
```

```
<ipython-input-3-ef32d6805f14> in <module>()
----> 1 v*A

/usr/lib/sagemath/local/lib/python2.7/site-packages/sage/structure/ele

/usr/lib/sagemath/local/lib/python2.7/site-packages/sage/structure/coe

TypeError: unsupported operand parent(s) for '*': 'Vector space of
                  dimension 4 over Rational Field' and 'Full MatrixSpace
                  of 2 by 4 dense matrices over Rational Field'
```

As in the earlier error message, some lines are truncated but the final line is wrapped twice to show it all. It says, in short, that the product of a size 4 vector with a size 2×4 matrix is not defined.

Of course you can multiply from the left by a vector if it has a size that matches the matrix.

```
sage: w = vector(QQ, [3, 5])
sage: w*A
(3, 19, 35, 57)
```

In practice you will sometimes see matrix-vector multiplications done with vectors on the left, and sometimes on the right. The textbook has the vector on the right, to have the product $H\vec{x}$ fit visually with the map application $h(x)$. *Sage* will do either, although it has something of a preference for the vector on the left (as we will see in Chapter 5).

Matrix-matrix product If the sizes match then *Sage* will multiply the matrices. Here is the product of a 2×2 matrix A and a 2×3 matrix B.

```
sage: A = matrix(QQ, [[2, 1], [4, 3]])
sage: B = matrix(QQ, [[5, 6, 7], [8, 9, 10]])
sage: A*B
[18 21 24]
[44 51 58]
```

Trying `B*A` gives an error `TypeError: unsupported operand parent(s) for '*'`, reflecting that the product operation in this order is undefined.

Same-sized square matrices have the product defined in either order.

```
sage: A = matrix(QQ, [[1, 2], [3, 4]])
sage: B = matrix(QQ, [[4, 5], [6, 7]])
sage: A*B
[16 19]
[36 43]
sage: B*A
[19 28]
[27 40]
```

They are different; matrix multiplication is not commutative.

```
sage: A*B == B*A
False
```

In fact, matrix multiplication is very non-commutative in that if you produce two $n\times n$ matrices at random then they almost surely don't commute. *Sage* lets us produce matrices at random.

```
sage: random_matrix(RDF, 3, min=-1, max=1)
[ -0.500967630574    0.399154213767    0.158623040807]
[  0.592352667129    0.805088498682  -0.0592516994557]
[ -0.450830529715   -0.671471002473   -0.164727579764]
sage: random_matrix(RDF, 3, min=-1, max=1)
[ 0.281077275057  -0.691741095475   0.367171520896]
[  0.46410058747   0.727739706971  -0.284325541413]
[  0.30924236249   0.808344132623   0.312388757539]
```

(Note the `RDF`. We prefer real number entries here because *Sage*'s `random_matrix` is more straightforward in this case than in the rational entry case.)

```
sage: number_commuting = 0
sage: for n in range(1000):
....:     A = random_matrix(RDF, 2, min=-1, max=1)
....:     B = random_matrix(RDF, 2, min=-1, max=1)
....:     if (A*B == B*A):
....:         number_commuting = number_commuting + 1
....:
sage: print "number commuting of 1000=", number_commuting
number commuting of 1000= 0
```

Inverse Recall that if A is nonsingular then its *inverse* A^{-1} is the matrix such that $AA^{-1} = A^{-1}A$ is the identity matrix.

```
sage: A = matrix(QQ, [[1, 3, 1], [2, 1, 0], [4, -1, 0]])
sage: A.is_singular()
False
```

For 2×2 matrix inverses we have a formula. In the book to do by-hand inverses for larger cases we write the original matrix next to the identity and then do Gauss-Jordan reduction.

```
sage: I = identity_matrix(3)
sage: B = A.augment(I, subdivide=True)
sage: B
[ 1  3  1| 1  0  0]
[ 2  1  0| 0  1  0]
[ 4 -1  0| 0  0  1]
sage: C = B.rref()
sage: C
[    1     0     0|    0   1/6   1/6]
[    0     1     0|    0   2/3  -1/3]
[    0     0     1|    1 -13/6   5/6]
```

The inverse is the resulting matrix on the right.

```
sage: A_inv = C.matrix_from_columns([3, 4, 5])
sage: A_inv
[    0   1/6   1/6]
[    0   2/3  -1/3]
[    1 -13/6   5/6]
sage: A*A_inv
[1 0 0]
[0 1 0]
```

```
[0  0  1]
sage: A_inv*A
[1  0  0]
[0  1  0]
[0  0  1]
```

This is an operation that *Sage* users do all the time so there is a standalone command.

```
sage: A_inv = A.inverse()
sage: A_inv
[     0    1/6    1/6]
[     0    2/3   -1/3]
[     1 -13/6    5/6]
```

One reason for finding the inverse is to make solving linear systems easier. These three systems

$$\begin{array}{rcl} x+3y+z=4 \\ 2x+y=4 \\ 4x-y=4 \end{array} \qquad \begin{array}{rcl} x+3y+z=2 \\ 2x+y=-1 \\ 4x-y=5 \end{array} \qquad \begin{array}{rcl} x+3y+z=1/2 \\ 2x+y=0 \\ 4x-y=12 \end{array}$$

share the matrix of coefficients but have different vectors on the right side. If you have first calculated the inverse of the matrix of coefficients then solving each system takes just a matrix-vector product.

```
sage: A = matrix(QQ, [[1, 3, 1], [2, 1, 0], [4, -1, 0]])
sage: A_inv = A.inverse()
sage: v1 = vector(QQ, [4, 4, 4])
sage: v2 = vector(QQ, [2, -1, 5])
sage: v3 = vector(QQ, [1/2, 0, 12])
sage: A_inv*v1
(4/3, 4/3, -4/3)
sage: A_inv*v2
(2/3, -7/3, 25/3)
sage: A_inv*v3
(2, -4, 21/2)
```

Running time

Since computers are fast and accurate they open up the possibility of solving problems that are quite large. Large linear algebra problems occur frequently in science and engineering. In this section we will suggest what limits there are to how big the problems can get, and still be solvable. (In this section we will use matrices with real entries because they are common in applications.)

One of the limits on just how large a problem we can do is how quickly the computer can give us the answer. Naturally computers take longer to perform operations on matrices that are larger but it may be that the time the program takes to compute the answer grows more quickly than does the size of the problem — for instance, it may be that when the size of the problem doubles then the time to do the job more than doubles.

The matrix inverse operation is a good illustration. This is an important operation; for instance, if we could do large matrix inverses quickly then we could quickly solve large linear systems, with just a matrix-vector product.

```
sage: A = matrix(RDF, [[1, 3, 1], [2, 1, 0], [4, -1, 0]])
sage: A
[ 1.0  3.0  1.0]
[ 2.0  1.0  0.0]
[ 4.0 -1.0  0.0]
sage: A.is_singular()
False
sage: timeit('A.inverse()')
625 loops, best of 3: 52.6 µs per loop
```

Sage's `timeit` makes a best guess about how long the operation ideally takes by running the command a number of times, because on any one time your computer may have been slowed down by a disk write or some other interruption.

The inverse operation took on the order of hundreds of microseconds. A single microsecond is 0.000 001 seconds. That's fast, but then A is only a 3×3 matrix.

And, A is a particular 3×3 matrix. You'd like to know how long it takes to invert a generic, or average, matrix. You could try finding the inverse of a random matrix.

```
sage: timeit('random_matrix(RDF, 3, min=-1, max=1).inverse()')
625 loops, best of 3: 78 µs per loop
sage: timeit('random_matrix(RDF, 3, min=-1, max=1).inverse()')
625 loops, best of 3: 77.6 µs per loop
sage: timeit('random_matrix(RDF, 3, min=-1, max=1).inverse()')
625 loops, best of 3: 77.8 µs per loop
```

Again this is of the order of hundreds of microseconds.

But this has the issue that we can't tell from it whether the time is spent generating the random matrix or finding the inverse. In addition, there is a subtler point: we also can't tell right away if this command generates many random matrices and finds each's inverse, or if it generates one random matrix and applies the inverse many times. The next code at least makes that point clear. For the sizes 3×3, 10×10, etc., if finds a single random matrix and then gets the time to compute its inverse.

```
sage: for size in [3, 10, 25, 50, 75, 100, 150, 200]:
....:     print "size=",size
....:     M = random_matrix(RR, size, min=-1, max=1)
....:     timeit('M.inverse()')
....:
size= 3
625 loops, best of 3: 125 µs per loop
size= 10
625 loops, best of 3: 940 µs per loop
size= 25
25 loops, best of 3: 12 ms per loop
size= 50
5 loops, best of 3: 92.4 ms per loop
size= 75
5 loops, best of 3: 308 ms per loop
size= 100
5 loops, best of 3: 727 ms per loop
size= 150
5 loops, best of 3: 2.45 s per loop
```

```
size= 200
5 loops, best of 3: 5.78 s per loop
```

Some of those times are in microseconds, some are in milliseconds, and some are in seconds. This table is consistently in seconds.

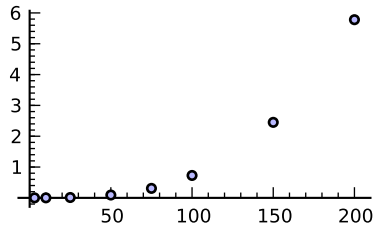
size	*seconds*
3	0.000 125
10	0.000 940
25	0.012
50	0.092 4
75	0.308
100	0.727
150	2.45
200	5.78

The time grows faster than the size. For instance, in going from size 25 to size 50 the time more than doubles: 0.0924/0.012 is 7.7. Similarly, increasing the size from 50 to 200 causes the time to increase by much more than a factor of four: $5.78/0.0924 \approx 62.55$.

To get a picture give *Sage* the data as a list of pairs.

```
sage: d = [(3, 0.000125), (10, 0.000940), (25, 0.012),
....:       (50, 0.0924), (75, 0.308), (100, 0.727),
....:       (150, 2.45), (200, 5.78)]
sage: g = scatter_plot(d)
sage: g.save("graphics/mat001.pdf")
```

(If you enter `scatter_plot(d)` at the prompt, that is, without saving it as `g`, then *Sage* will pop up a window with the graphic.)[1]



The graph dramatizes that the ratio time/size is not constant since the data clearly does not lie on a line.

Here is some more data. The times are big enough that this computer had to run overnight.

```
sage: for size in [500, 750, 1000]:
....:           print "size=",size
....:       M = random_matrix(RR, size, min=-1, max=1)
....:       timeit('M.inverse()')
....:
```

[1] The graphics in this manual are generated using more drawing options than appear in the output block. For instance, the scatter plot here came from `g = scatter_plot(d, markersize=10, facecolor='#b9b9ff')` and was saved in a file with `g.save("graphics/mat001.pdf", figsize=[2.25,1.5], axes_pad=0.05, fontsize=7)`. We shall omit much of this code about decoration as clutter. See the *Sage* manual for `plot` options.

```
size= 500
5 loops, best of 3: 89.2 s per loop
size= 750
5 loops, best of 3: 299 s per loop
size= 1000
5 loops, best of 3: 705 s per loop
```

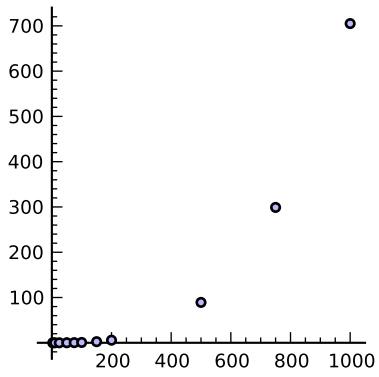
Again the table is a neater way to present the data.

size	*seconds*
500	89.2
750	299.
1000	705.

Get a graph by tacking the new data onto the existing data.

```
sage: d = d + [(500, 89.2), (750, 299), (1000, 705)]
sage: g = scatter_plot(d)
sage: g.save("graphics/mat002.pdf")
```

The result is this graphic.



Note that the two graphs have different scales; if you generated this graph with the same vertical scale as the prior one then the data would fall off the top of the page.

So a practical limit to the size of a problem that we can solve with this matrix inverse operation comes from the fact that the graph above is not a line. The time required grows much faster than the size, and just gets too large.

A major effort in Computer Science is to find fast algorithms to do practical tasks. Many people have worked on tasks in Linear Algebra in particular, such as finding the inverse of a matrix, because they are so common in applications.

Maps

We've used *Sage* to define vector spaces. Next we explore operations that you can do on vector spaces.

Left/right

Sage represents linear maps differently than the book does. An example is in representing the application of a linear map to a member of a vector space $t(\vec{v})$. With this function $t\colon \mathbb{R}^2 \to \mathbb{R}^3$ and this element of the domain

$$t(\begin{pmatrix} a \\ b \end{pmatrix}) = \begin{pmatrix} a+b \\ a-b \\ b \end{pmatrix} \qquad \vec{v} = \begin{pmatrix} 1 \\ 3 \end{pmatrix}$$

the map application gives this output.

$$t(\begin{pmatrix} 1 \\ 3 \end{pmatrix}) = \begin{pmatrix} 4 \\ -2 \\ 3 \end{pmatrix}$$

To represent the map application we first fix bases. In this example we use the canonical bases $E_2 \subset \mathbb{R}^2$ and $E_3 \subset \mathbb{R}^3$. With respect to the bases, the book finds a matrix $T = \mathrm{Rep}_{E_2,E_3}(t)$ and a column vector $\vec{w} = \mathrm{Rep}_{E_2}(\vec{v})$, and represents the map application $t(\vec{v})$ with the matrix-vector product $T\vec{w}$.

$$\begin{pmatrix} 1 & 1 \\ 1 & -1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 3 \end{pmatrix} = \begin{pmatrix} 4 \\ -2 \\ 3 \end{pmatrix}$$

That is, the book is write-right: its notation puts the vector on the right of the matrix.

However, this choice is a matter of taste and many authors instead use a row vector that multiplies a matrix from the left. *Sage* is in this camp and represents the map application in this way.

$$\begin{pmatrix} 1 & 3 \end{pmatrix} \begin{pmatrix} 1 & 1 & 0 \\ 1 & -1 & 1 \end{pmatrix} = \begin{pmatrix} 4 & -2 & 3 \end{pmatrix}$$

Obviously the difference is cosmetic but can cause confusion. Rather than quarrel with the tool, in this manual we will do it *Sage*'s way. The translation is that, compared to the book's $T\vec{w}$, *Sage* prefers the transpose $(T\vec{w})^{\mathsf{T}} = \vec{w}^{\mathsf{T}}\, T^{\mathsf{T}}$.

Defining

We will see two different ways to define a linear transformation.

Symbolically We first define a map that takes two inputs and returns three outputs.

```
sage: a, b = var('a, b')
sage: T_symbolic(a, b) = [a+b, a-b, b]
sage: T_symbolic
(a, b) |--> (a + b, a - b, b)
```

We have not yet defined a domain and codomain so this not a function — instead it is a prototype for a function. Make an instance of a function by applying $T_{symbolic}$ on a particular domain and codomain.

```
sage: T = linear_transformation(QQ^2, QQ^3, T_symbolic)
sage: T
Vector space morphism represented by the matrix:
[ 1  1  0]
[ 1 -1  1]
Domain: Vector space of dimension 2 over Rational Field
Codomain: Vector space of dimension 3 over Rational Field
```

Note the left/right issue again: *Sage*'s matrix is the transpose of the matrix that the book would use.

Evaluating this function on a member of the domain gives a member of the codomain.

```
sage: v = vector(QQ, [1, 3])
sage: v
(1, 3)
sage: T(v)
(4, -2, 3)
```

Sage can compute the interesting things about the transformation. Here it finds the null space and range space, using the equivalent terms *kernel* and *image*.

```
sage: T.kernel()
Vector space of degree 2 and dimension 0 over Rational Field
Basis matrix:
[]
sage: T.image()
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[   1    0  1/2]
[   0    1 -1/2]
```

The null space is the trivial subspace of the domain so its basis is empty, with dimension 0. Therefore T is one-to-one.

The range space has a two-vector basis. This fits with the theorem that the dimension of the null space plus the dimension of the range space equals to the dimension of the domain.

For contrast consider a map that is not one-to-one.

```
sage: S_symbolic(a, b) = [a+2*b, a+2*b]
sage: S_symbolic
```

```
(a, b) |--> (a + 2*b, a + 2*b)
sage: S = linear_transformation(QQ^2, QQ^2, S_symbolic)
sage: S
Vector space morphism represented by the matrix:
[1 1]
[2 2]
Domain: Vector space of dimension 2 over Rational Field
Codomain: Vector space of dimension 2 over Rational Field
sage: v = vector(QQ, [1, 3])
sage: S(v)
(7, 7)
```

This map is not one-to-one since the input $(a, b) = (2, 0)$ gives the same result as $(a, b) = (0, 1)$.

```
sage: S(vector(QQ, [2, 0]))
(2, 2)
sage: S(vector(QQ, [0, 1]))
(2, 2)
```

Another way to tell that the map is not one-to-one is to look at the null space.

```
sage: S.kernel()
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[   1 -1/2]
sage: S.image()
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[1 1]
```

The null space has nonzero dimension, namely it has dimension 1, so *Sage* agrees that the map is not one-to-one.

Without looking at the range space we know that its dimension must be 1 because the dimensions of the null and range spaces add to the dimension of the domain. *Sage* confirms our calculation.

Via matrices We can define a transformation by specifying the matrix representing its action.

```
sage: M = matrix(QQ, [[1, 2], [3, 4], [5, 6]])
sage: M
[1 2]
[3 4]
[5 6]
sage: m = linear_transformation(M)
sage: m
Vector space morphism represented by the matrix:
[1 2]
[3 4]
[5 6]
Domain: Vector space of dimension 3 over Rational Field
Codomain: Vector space of dimension 2 over Rational Field
```

Note again that *Sage* prefers the representation where the vector multiplies from the left.

```
sage: v = vector(QQ, [7, 8, 9])
sage: v
(7, 8, 9)
sage: m(v)
(76, 100)
```

Sage has done this calculation.

$$\begin{pmatrix}7 & 8 & 9\end{pmatrix}\begin{pmatrix}1 & 2\\ 3 & 4\\ 5 & 6\end{pmatrix}=\begin{pmatrix}76 & 100\end{pmatrix}$$

If you have a matrix intended for a vector-on-the-right calculation (as in the book) then *Sage* will make the necessary adaptation.

```
sage: N = matrix(QQ, [[1, 3, 5], [2, 4, 6]])
sage: n = linear_transformation(N, side='right')
sage: n
Vector space morphism represented by the matrix:
[1 2]
[3 4]
[5 6]
Domain: Vector space of dimension 3 over Rational Field
Codomain: Vector space of dimension 2 over Rational Field
sage: v = vector(QQ, [7, 8, 9])
sage: v
(7, 8, 9)
sage: n(v)
(76, 100)
```

Although we gave it a `side='right'` option, the matrix that *Sage* shows by default is for `side='left'`.

Despite that we specified them differently, these two transformations are the same.

```
sage: M = matrix(QQ, [[1, 2], [3, 4], [5, 6]])
sage: m = linear_transformation(M)
sage: N = matrix(QQ, [[1, 3, 5], [2, 4, 6]])
sage: n = linear_transformation(N, side='right')
sage: m == n
True
```

We can ask the same questions of linear transformations created from matrices that we asked of linear transformations created from functions.

```
sage: M = matrix(QQ, [[1, 2], [3, 4], [5, 6]])
sage: m = linear_transformation(M)
sage: m.kernel()
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[ 1 -2  1]
```

The null space of m is not the trivial subspace of $\mathbb{R}^3$ and so this function is not one-to-one. The domain has dimension 3 and the null space has dimension 1 and so the range space is a dimension 2 subspace of $\mathbb{R}^2$.

```
sage: m.image()
Vector space of degree 2 and dimension 2 over Rational Field
Basis matrix:
[1 0]
[0 1]
sage: m.image() == QQ^2
True
```

Sage lets us have the matrix represent a transformation involving spaces with nonstandard bases.

```
sage: M = matrix(QQ, [[1, 2], [3, 4]])
sage: delta_1 = vector(QQ, [1, -1])
sage: delta_2 = vector(QQ, [1, 1])
sage: domain_basis = [delta_1, delta_2]
sage: domain_basis
[(1, -1), (1, 1)]
sage: D = (QQ^2).subspace_with_basis(domain_basis)
sage: gamma_1 = vector(QQ, [2, 0])
sage: gamma_2 = vector(QQ, [0, 3])
sage: codomain_basis = [gamma_1, gamma_2]
sage: codomain_basis
[(2, 0), (0, 3)]
sage: C = (QQ^2).subspace_with_basis(codomain_basis)
sage: m = linear_transformation(D, C, M)
sage: m
Vector space morphism represented by the matrix:
[1 2]
[3 4]
Domain: Vector space of degree 2 and dimension 2 over Rational Field
User basis matrix:
[ 1 -1]
[ 1  1]
Codomain: Vector space of degree 2 and dimension 2 over Rational Field
User basis matrix:
[2 0]
[0 3]
sage: m(vector(QQ, [1, 0]))
(4, 9)
```

Sage has calculated that

$$\begin{pmatrix}1\\0\end{pmatrix} = (1/2)\begin{pmatrix}1\\-1\end{pmatrix} + (1/2)\begin{pmatrix}1\\1\end{pmatrix} \quad \text{so} \quad \text{Rep}_{domain\_basis}(\begin{pmatrix}1\\0\end{pmatrix}) = \begin{pmatrix}1/2\\1/2\end{pmatrix}$$

and then computed this.

$$\text{Rep}_{codomain\_basis}(m(\vec{v})) = (1/2 \;\; 1/2)\begin{pmatrix}1 & 2\\3 & 4\end{pmatrix} = (2 \;\; 3) \quad \text{so} \quad m(\vec{v}) = 2\begin{pmatrix}2\\0\end{pmatrix} + 3\begin{pmatrix}0\\3\end{pmatrix} = \begin{pmatrix}4\\9\end{pmatrix}$$

Operations

Fix some vector space domain D and codomain C and consider the set of all linear transformations between them. This set has some natural operations, including addition and scalar multiplication. *Sage* can work with those operations.

Addition Recall that matrix addition is defined so that the representation of the sum of two linear transformations is the matrix sum of the representatives. *Sage* can illustrate.

```
sage: M = matrix(QQ, [[1, 2], [3, 4]])
sage: m = linear_transformation(QQ^2, QQ^2, M)
sage: m
Vector space morphism represented by the matrix:
[1 2]
[3 4]
Domain: Vector space of dimension 2 over Rational Field
Codomain: Vector space of dimension 2 over Rational Field
sage: N = matrix(QQ, [[5, -1], [0, 7]])
sage: n = linear_transformation(QQ^2, QQ^2, N)
sage: n
Vector space morphism represented by the matrix:
[ 5 -1]
[ 0  7]
Domain: Vector space of dimension 2 over Rational Field
Codomain: Vector space of dimension 2 over Rational Field
sage: m+n
Vector space morphism represented by the matrix:
[ 6  1]
[ 3 11]
Domain: Vector space of dimension 2 over Rational Field
Codomain: Vector space of dimension 2 over Rational Field
sage: M+N
[ 6  1]
[ 3 11]
```

Similarly, linear map scalar multiplication is reflected by matrix scalar multiplication.

```
sage: m*3
Vector space morphism represented by the matrix:
[ 3  6]
[ 9 12]
Domain: Vector space of dimension 2 over Rational Field
Codomain: Vector space of dimension 2 over Rational Field
sage: M*3
[ 3  6]
[ 9 12]
```

Composition The composition of linear maps gives rise to matrix multiplication. *Sage* uses the star $*$ to denote composition of linear maps.

```
sage: M = matrix(QQ, [[1, 2], [3, 4]])
sage: m = linear_transformation(QQ^2, QQ^2, M)
```

```
sage: m
Vector space morphism represented by the matrix:
[1 2]
[3 4]
Domain: Vector space of dimension 2 over Rational Field
Codomain: Vector space of dimension 2 over Rational Field
sage: N = matrix(QQ, [[5, -1], [0, 7]])
sage: n = linear_transformation(QQ^2, QQ^2, N)
sage: n
Vector space morphism represented by the matrix:
[ 5 -1]
[ 0  7]
Domain: Vector space of dimension 2 over Rational Field
Codomain: Vector space of dimension 2 over Rational Field
sage: m*n
Vector space morphism represented by the matrix:
[ 2  6]
[21 28]
Domain: Vector space of dimension 2 over Rational Field
Codomain: Vector space of dimension 2 over Rational Field
```

Note: there is a left/right issue here. As the book emphasizes, matrix multiplication is about representing the composition of the maps. The composition $m \circ n$ is the map $\vec{v} \mapsto m(n(\vec{v}))$, with n applied first. We can walk through the calculation of applying n first and then m

$$\begin{pmatrix}1\\2\end{pmatrix} \overset{n}{\longmapsto} \begin{pmatrix}5\\13\end{pmatrix} \overset{m}{\longmapsto} \begin{pmatrix}44\\62\end{pmatrix}$$

via these matrix multiplications.

$$\begin{pmatrix}1 & 2\end{pmatrix}\begin{pmatrix}5 & -1\\0 & 7\end{pmatrix} = \begin{pmatrix}5 & 13\end{pmatrix} \quad \text{followed by} \quad \begin{pmatrix}5 & 13\end{pmatrix}\begin{pmatrix}1 & 2\\3 & 4\end{pmatrix} = \begin{pmatrix}44 & 62\end{pmatrix}$$

Sage prefers the representing vector on the left so to be the first one done, N must come left-most: $m \circ n$ is represented by NM. Here *Sage* does the map application.

```
sage: M = matrix(QQ, [[1, 2], [3, 4]])
sage: N = matrix(QQ, [[5, -1], [0, 7]])
sage: M*N
[ 5 13]
[15 25]
sage: N*M
[ 2  6]
[21 28]
sage: m = linear_transformation(QQ^2, QQ^2, M)
sage: n = linear_transformation(QQ^2, QQ^2, N)
sage: t = m*n
sage: t
Vector space morphism represented by the matrix:
[ 2  6]
[21 28]
Domain: Vector space of dimension 2 over Rational Field
Codomain: Vector space of dimension 2 over Rational Field
```

```
sage: v = vector(QQ, [1, 2])
sage: t(v)
(44, 62)
```

Singular Value Decomposition

Recall that a line through the origin in $\mathbb{R}^n$ is $\{r \cdot \vec{v} \mid r \in \mathbb{R}\}$. One of the defining properties of a linear map is that $h(r \cdot \vec{v}) = r \cdot h(\vec{v})$. So the action of h on any line through the origin is determined by the action of h on any nonzero vector in that line.

For instance, consider the line $y = 2x$ in the plane,

$$\{r \cdot \begin{pmatrix} 1 \\ 2 \end{pmatrix} \mid r \in \mathbb{R}\}$$

suppose that $t\colon \mathbb{R}^2 \to \mathbb{R}^2$ is represented by the matrix

$$\mathrm{Rep}_{E_2,E_2}(t) = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

then here is the effect of t on one of the line's vectors (calculating with the vector on the left $\vec{v}T$).[1]

$$\vec{v} = \begin{pmatrix} 1 \\ 2 \end{pmatrix} \overset{t}{\longmapsto} \begin{pmatrix} 7 \\ 10 \end{pmatrix}$$

The point is that the scalar multiplication property in the definition of linear map imposes a simple uniformity on its action on lines through the origin: the map t has twice the effect on $2\vec{v}$, three times the effect on $3\vec{v}$, etc.

$$\begin{pmatrix} 2 \\ 4 \end{pmatrix} \overset{t}{\longmapsto} \begin{pmatrix} 14 \\ 20 \end{pmatrix} \qquad \begin{pmatrix} -3 \\ -6 \end{pmatrix} \overset{t}{\longmapsto} \begin{pmatrix} -21 \\ -30 \end{pmatrix} \qquad \begin{pmatrix} r \\ 2r \end{pmatrix} \overset{t}{\longmapsto} \begin{pmatrix} 7r \\ 10r \end{pmatrix}$$

The action of of a linear map on any vector in a line through the origin is determined by the action of that map on any single nonzero vector in that line.

Unit circle

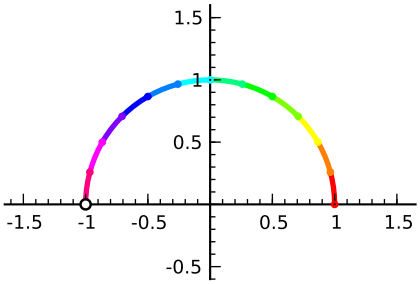
Consider transformations of the plane $\mathbb{R}^2$. Because of $t(r \cdot \vec{v}) = r \cdot t(\vec{v})$, one way to describe a transformation's action is to pick a set containing one nonzero vector from each line through the origin and describe where the transformation maps those elements.

A natural set that contains one nonzero element from each line through the origin is the upper half unit circle.

[1] Recall that *Sage* prefers to do matrix-vector multiplication as $\vec{v}M$ (here $\vec{v}$ is a row vector). Translate to the book's convention by transposing, $M^T\vec{v}^T$. See the discussion on page 43.

```
sage: runfile("plot_action.sage")
sage: p = plot_circle_action(1,0,0,1)
sage: p.set_axes_range(-1.5, 1.5, -0.5, 1.5)
sage: p.save("graphics/svd000.pdf")
```

$$U = \{ \begin{pmatrix} x \\ y \end{pmatrix} \mid x = \cos(t),\ y = \sin(t),\ 0 \leqslant t < \pi \}$$



The above graph came from using a routine that draws the effect of a transformation on the unit circle and then specifying the identity transformation. Its source code is at the end of this chapter but in short `plot_circle_action(a,b,c,d)` multiplies points on the unit circle by this matrix.

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

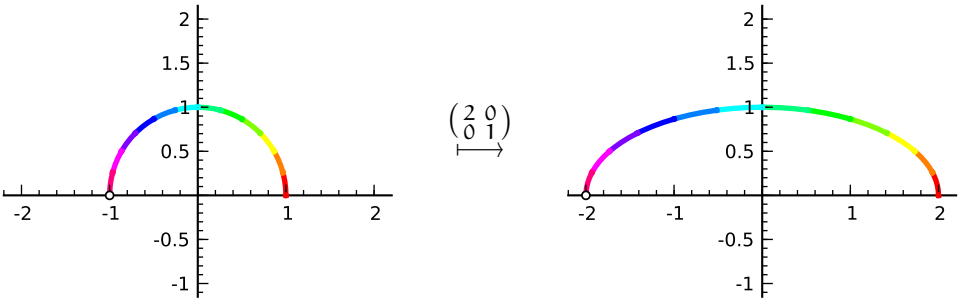
(The reason for the colors is given below.)

Here is the effect of the transformation

$$\begin{pmatrix} x \\ y \end{pmatrix} \mapsto \begin{pmatrix} 2x \\ y \end{pmatrix}$$

that doubles the x component of input vectors. It shows before and after, the upper half circle domain and the output codomain.

```
sage: p = plot_circle_action(2,0,0,1)
```

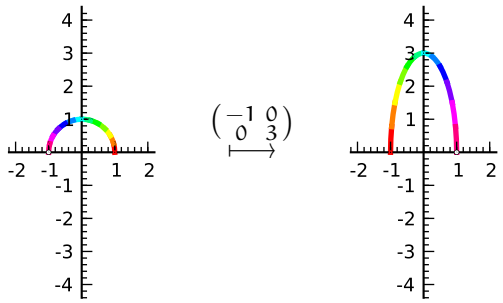


The next before and after picture shows the effect of the transformation

$$\begin{pmatrix} x \\ y \end{pmatrix} \mapsto \begin{pmatrix} -x \\ 3y \end{pmatrix}$$

that triples the y component and multiplies the x component by -1.

```
sage: p = plot_circle_action(-1,0,0,3)
```



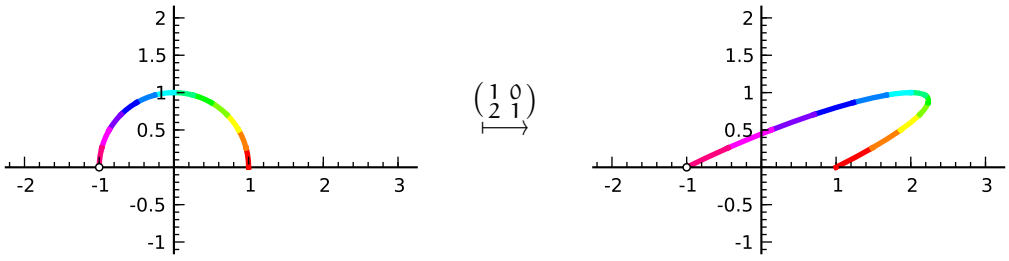
Now the colors come in. The input circle moves counterclockwise from red to orange, then to green, blue, indigo, and finally to violet. But the output does the opposite: to move from red to violet you move clockwise. This transformation changes the *orientation* (or *sense*) of the curve.

The next transformation is a skew.

$$\begin{pmatrix} x \\ y \end{pmatrix} \mapsto \begin{pmatrix} x + 2y \\ y \end{pmatrix}$$

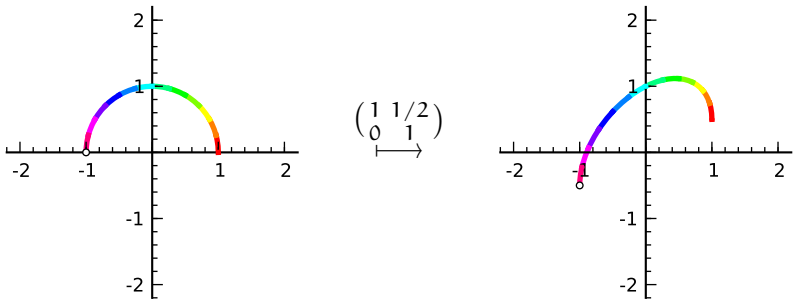
The output's first component is affected by the input vector's distance from the y-axis.

```
sage: p = plot_circle_action(1,0,2,1)
```



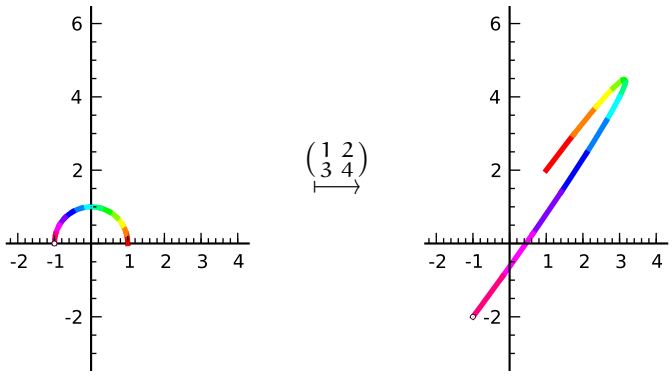
Here is another skew. In this case the output's second component is affected by the input's distance from the x-axis.

```
sage: p = plot_circle_action(1,1/2,0,1)
```



And here is a generic transformation. It changes orientation also.

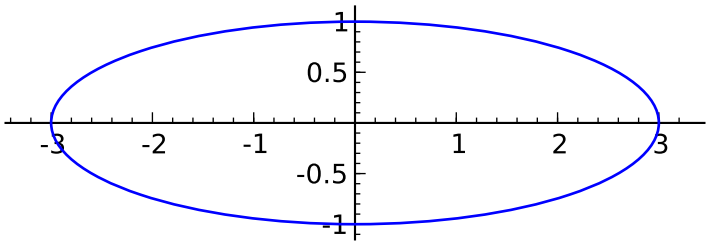
```
sage: p = plot_circle_action(1,2,3,4)
```



SVD

The above pictures show the unit circle mapping to ellipses. Recall that in $\mathbb{R}^2$ an ellipse has a *major axis*, the longer one, and a *minor axis*.[1] Write σ_1 for the length of the semi-major axis, the distance from the center to the furthest-away point on the ellipse, and write σ_2 for the length of the semi-minor axis.

```
sage: sigma_1=3
sage: sigma_2=1
sage: E = ellipse((0,0), sigma_1, sigma_2)
sage: E.save("graphics/svd100.pdf",figsize=4)
```



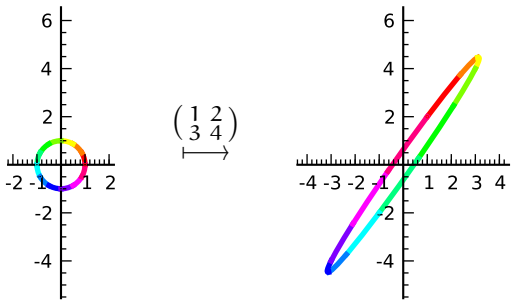
In an ellipse the two axes are orthogonal. In the above graph the major axis lies along the x-axis while the minor axis lies along the y-axis.

Under any linear map $t\colon \mathbb{R}^n \to \mathbb{R}^m$ the unit sphere maps to a hyperellipse. This is a version of the *Singular Value Decomposition* of matrices: for any linear map $t\colon \mathbb{R}^m \to \mathbb{R}^n$ there are bases $B = \langle \vec{\beta}_1, \ldots, \vec{\beta}_m \rangle$ for the domain and $D = \langle \vec{\delta}_1, \ldots, \vec{\delta}_n \rangle$ for the codomain such that $t(\vec{\beta}_i) = \sigma_i \vec{\delta}_i$, where the *singular values* σ_i are scalars. The next section sketches a proof but we first illustrate this result by using an example matrix. *Sage* will find the two bases B and D and will picture how the vectors $\vec{\beta}_i$ are mapped to the $\sigma_i \vec{\delta}_i$.

So consider again the generic matrix. Here is its action again, this time shown on a full circle.

```
sage: p = plot_circle_action(1,2,3,4,full_circle=True)
```

[1] If the two axes have the same length then the ellipse is a circle. If one axis has length zero then the ellipse is a line segment and if both have length zero then it is a point.



(*)

Sage will find the SVD of this example matrix.

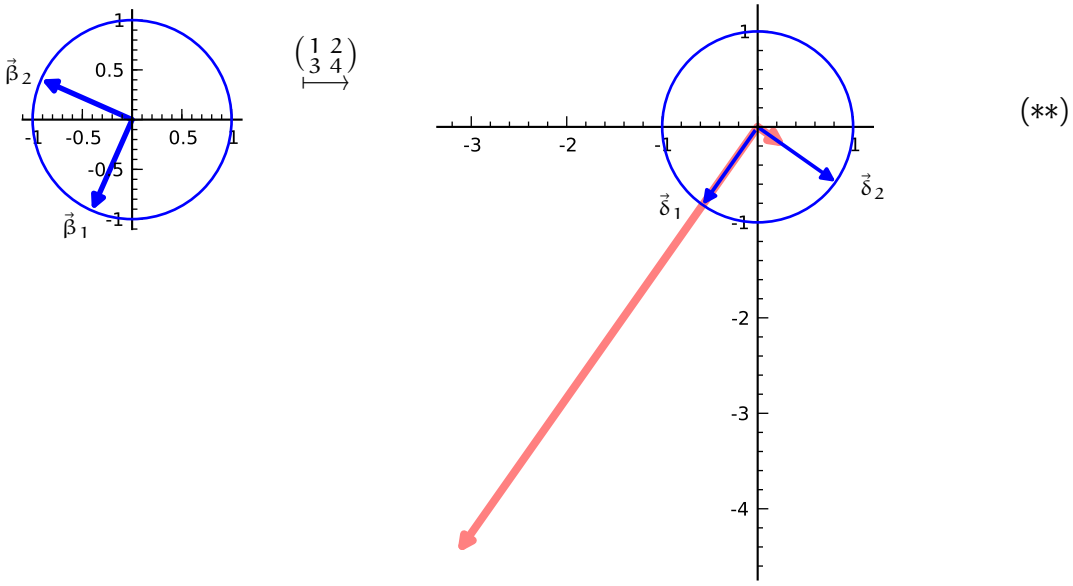
```
sage: M = matrix(RDF, [[1, 2], [3, 4]])
sage: U,Sigma,V = M.SVD()
sage: U
[-0.404553584834 -0.914514295677]
[-0.914514295677  0.404553584834]
sage: Sigma
[ 5.46498570422              0.0]
[            0.0 0.365966190626]
sage: V
[-0.576048436766  0.81741556047]
[ -0.81741556047 -0.576048436766]
sage: U*Sigma*(V.transpose())
[1.0 2.0]
[3.0 4.0]
```

The Singular Value Decomposition has M as the product of three matrices, $U\Sigma V^T$. The basis vectors $\vec{\beta}_1$, $\vec{\beta}_2$, $\vec{\delta}_1$, and $\vec{\delta}_2$ are the columns of U and V. The singular values are the diagonal entries of Σ. *Sage* will plot the effect of the transformation on the basis vectors for the domain so we can compare those with the basis vectors for the codomain.

```
sage: beta_1 = vector(RDF, [U[0][0], U[1][0]])
sage: beta_2 = vector(RDF, [U[0][1], U[1][1]])
sage: delta_1 = vector(RDF, [V[0][0], V[1][0]])
sage: delta_2 = vector(RDF, [V[0][1], V[1][1]])
sage: P = C + plot(beta_1) + plot(beta_2)
sage: P.save("graphics/svd102a.pdf",figsize=2)
sage: image_color=Color(1,0.5,0.5)    # color for t(beta_1), t(beta_2)
sage: Q = C + plot(beta_1*M, width=3, color=image_color)
sage: Q = Q + plot(delta_1,width=1.4, color='blue')
sage: Q = Q + plot(beta_2*M,width=3, color=image_color)
sage: Q = Q + plot(delta_2,width=1.4, color='blue')
sage: Q.save("graphics/svd102b.pdf")
```

In the picture below the domain's blue $\vec{\beta}$'s on the left map to the codomain's light red $t(\vec{\beta})$'s on the right. Also on the right, in blue, are the $\vec{\delta}$'s. The red $t(\vec{\beta}_1)$ does look to be about 5.5 times

$\vec{\delta}_1$, and $t(\vec{\beta}_2)$ does look something like 0.4 times $\vec{\delta}_2$.



Note also that the two bases are *orthonormal*—the unit circles help us see that the bases are comprised of unit vectors and further, the two members of each basis are orthogonal.

Compare this diagram to the one before it labeled (∗), which shows the effect of the matrix on the unit circle. We used the whole circle in (∗) to spotlight the ellipse and to make clearer that in (∗∗) the longest red vector is a semi-major axis of that ellipse and the shortest red vector is a semi-minor axis.

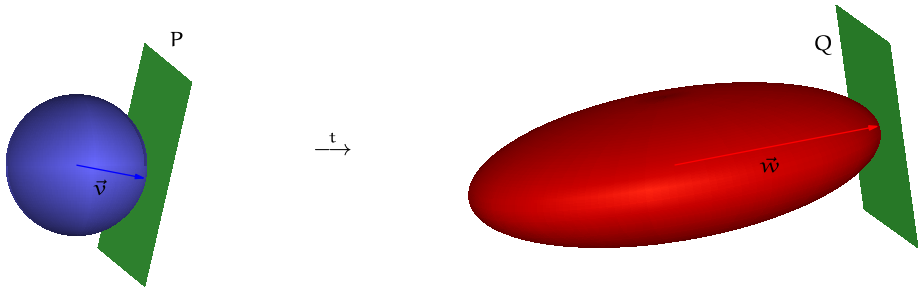
Proof sketch

This argument, from Blank et al. [1989], is a sketch because it uses results that a typical reader has only seen in a less general version and because it relies on material from the book that is optional. In addition, we'll consider only the case of a nonsingular matrix and map; it shows the main idea, which is the point of a sketch.

Consider an $n \times n$ matrix T that is nonsingular, and the nonsingular transformation $t: \mathbb{R}^n \to \mathbb{R}^n$ represented by T with respect to the standard bases.

Recall Calculus I's Extreme Value Theorem: for a continuous function f, if a subset $D \subset \mathbb{R}$ of the real line is closed and bounded then its image $f(D) = \{f(d) \mid d \in D\}$ is also closed and bounded (see Wikipedia [2012a]). A generalization of that result gives that because the unit sphere in $\mathbb{R}^n$ is closed and bounded then its image under t is closed and bounded. Although we won't prove this, the image is an ellipsoid so we will call it that.

Because this ellipsoid is closed and bounded it has a point furthest from the origin (it may have more than one but we just pick one). Let $\vec{w}$ be a vector extending from the origin to that furthest point. Let $\vec{v}$ be the member of the unit sphere that maps to $\vec{w}$. Let P be the plane tangent to the sphere at the endpoint of $\vec{v}$. Let Q be the image of P under t. Since t is one-to-one, Q touches the ellipsoid only at $\vec{w}$.



The picture illustrates that if we slide P along the vector $\vec{v}$ to the origin then we can recognize it as the subspace of $\mathbb{R}^n$ of vectors perpendicular to $\vec{v}$. This subspace has dimension $n-1$. We will argue in the next paragraph that similarly Q contains only vectors perpendicular to $\vec{w}$. With that we will have what we need for an argument by induction: we start constructing the bases B and D by taking $\vec{\beta}_1$ to be $\vec{v}$, taking σ_1 to be the length $\|\vec{w}\|$, and taking $\vec{\delta}_1$ to be $\vec{w}/\|\vec{w}\|$. Then induction proceeds by taking the restriction of t to P.

Therefore consider Q. The plane that touches the ellipsoid only at $\vec{w}$ is unique since if there were another then its inverse image under t would be a second plane, besides P, that touches the sphere only at $\vec{v}$, which is impossible (since it is a sphere). To see that Q is perpendicular to $\vec{w}$ consider a sphere in the codomain centered at the origin whose radius is $\|\vec{w}\|$. This sphere has a plane tangent at the endpoint of $\vec{w}$ that is perpendicular to $\vec{w}$. Because $\vec{w}$ ends at a point on the ellipsoid furthest from the origin, the ellipsoid is entirely contained in this sphere, so its tangent plane touches the ellipsoid only at $\vec{w}$. Therefore, from the second sentence of this paragraph, this tangent plane is Q. That ends the argument.

Matrix factorization

We can express those geometric ideas in an algebraic form (for a proof see Trefethen and Bau [1997]).

The *singular value decomposition* of an $m\times n$ matrix A is a factorization $A = U\Sigma V^T$. The $m\times n$ matrix Σ is all zeroes except for diagonal entries, the singular values, $\sigma_1 \geqslant \sigma_2 \geqslant \cdots \geqslant \sigma_r > 0$ where r is the rank of A. The $m\times m$ matrix U and the $n\times n$ matrix V are unitary, meaning that their columns form an orthogonal basis of unit vectors, the left and right *singular vectors* for A, respectively.

```
sage: M = matrix(RDF, [[0, 1, 2], [3, 4, 5]])
sage: U,Sigma,V = M.SVD()
sage: U
[-0.274721127897 -0.961523947641]
[-0.961523947641  0.274721127897]
sage: Sigma
[7.34846922835            0.0            0.0]
[           0.0            1.0            0.0]
sage: V
[-0.392540507864  0.824163383692  0.408248290464]
[-0.560772154092  0.137360563949 -0.816496580928]
[ -0.72900380032 -0.549442255795  0.408248290464]
```

The product $U\Sigma V^T$ simplifies. To see how, consider the case where all three matrices are 2×2.

Write $\vec{u}_1$, $\vec{u}_2$ for the columns of U and $\vec{v}_1$, $\vec{v}_2$ for the columns of V, so that the rows of V^{T} are $\vec{v}_1^{\mathsf{T}}$ and $\vec{v}_2^{\mathsf{T}}$.

$$\begin{aligned}
U\Sigma V^{\mathsf{T}} &= \begin{pmatrix}\vec{u}_1 & \vec{u}_2\end{pmatrix}\begin{pmatrix}\sigma_1 & 0\\ 0 & \sigma_2\end{pmatrix}\begin{pmatrix}\vec{v}_1^{\mathsf{T}}\\ \vec{v}_2^{\mathsf{T}}\end{pmatrix}\\
&= \begin{pmatrix}\vec{u}_1 & \vec{u}_2\end{pmatrix}\left[\begin{pmatrix}\sigma_1 & 0\\ 0 & 0\end{pmatrix}+\begin{pmatrix}0 & 0\\ 0 & \sigma_2\end{pmatrix}\right]\begin{pmatrix}\vec{v}_1^{\mathsf{T}}\\ \vec{v}_2^{\mathsf{T}}\end{pmatrix}\\
&= \begin{pmatrix}\vec{u}_1 & \vec{u}_2\end{pmatrix}\begin{pmatrix}\sigma_1 & 0\\ 0 & 0\end{pmatrix}\begin{pmatrix}\vec{v}_1^{\mathsf{T}}\\ \vec{v}_2^{\mathsf{T}}\end{pmatrix}+\begin{pmatrix}\vec{u}_1 & \vec{u}_2\end{pmatrix}\begin{pmatrix}0 & 0\\ 0 & \sigma_2\end{pmatrix}\begin{pmatrix}\vec{v}_1^{\mathsf{T}}\\ \vec{v}_2^{\mathsf{T}}\end{pmatrix}\\
&= \sigma_1\cdot\begin{pmatrix}\vec{u}_1 & \vec{u}_2\end{pmatrix}\begin{pmatrix}1 & 0\\ 0 & 0\end{pmatrix}\begin{pmatrix}\vec{v}_1^{\mathsf{T}}\\ \vec{v}_2^{\mathsf{T}}\end{pmatrix}+\sigma_2\cdot\begin{pmatrix}\vec{u}_1 & \vec{u}_2\end{pmatrix}\begin{pmatrix}0 & 0\\ 0 & 1\end{pmatrix}\begin{pmatrix}\vec{v}_1^{\mathsf{T}}\\ \vec{v}_2^{\mathsf{T}}\end{pmatrix} \qquad (***)
\end{aligned}$$

In the first term, right multiplication by the $1,1$ unit matrix picks out the first column of U, and left multiplication by the $1,1$ unit matrix picks out first row of V so those are the only parts that remain after the product. In short, we get this.

$$\begin{pmatrix}u_{1,1} & u_{1,2}\\ u_{2,1} & u_{2,2}\end{pmatrix}\begin{pmatrix}1 & 0\\ 0 & 0\end{pmatrix}\begin{pmatrix}v_{1,1} & v_{2,1}\\ v_{1,2} & v_{2,2}\end{pmatrix}=\begin{pmatrix}u_{1,1}v_{1,1} & u_{1,1}v_{2,1}\\ u_{2,1}v_{1,1} & u_{2,1}v_{2,1}\end{pmatrix}=\begin{pmatrix}u_{1,1}\\ u_{2,1}\end{pmatrix}\begin{pmatrix}v_{1,1} & v_{2,1}\end{pmatrix}=\vec{u}_1\vec{v}_1^{\mathsf{T}}$$

The second term simplifies in the same way.

$$\begin{pmatrix}u_{1,1} & u_{1,2}\\ u_{2,1} & u_{2,2}\end{pmatrix}\begin{pmatrix}0 & 0\\ 0 & 1\end{pmatrix}\begin{pmatrix}v_{1,1} & v_{2,1}\\ v_{1,2} & v_{2,2}\end{pmatrix}=\begin{pmatrix}u_{1,2}v_{1,2} & u_{1,2}v_{2,2}\\ u_{2,2}v_{1,2} & u_{2,2}v_{2,2}\end{pmatrix}=\vec{u}_2\vec{v}_2^{\mathsf{T}}$$

Thus, equation $(***)$ simplifies to $U\Sigma V^{\mathsf{T}}=\sigma_1\cdot\vec{u}_1\vec{v}_1^{\mathsf{T}}+\sigma_2\cdot\vec{u}_2\vec{v}_2^{\mathsf{T}}$. Cases other than 2×2 work the same way.

Application: data compression

We can write any matrix as a sum $M=\sigma_1\cdot\vec{u}_1\vec{v}_1^{\mathsf{T}}+\sigma_2\cdot\vec{u}_2\vec{v}_2^{\mathsf{T}}+\cdots$ where the vectors have unit size and the σ_i's decrease in size.

Suppose that the matrix is $n\times n$. To work with it — for example, to store it in a computer's memory or to transmit it over the Internet — we must work with n^2 real numbers (that is, floating point numbers). For instance, if $n=500$ then we have $50^2=250\,000$ reals. If we instead express the matrix as a sum then each term requires n real numbers for $\vec{u}_i$, another n reals for $\vec{v}_i$, and one more real for σ_i. So keeping all the terms would require $500\cdot(2\cdot 500+1)=500\,500$ reals, which is twice the data of the original matrix.

But keeping only some of the terms, say the first 50 of them, would be a savings: $50\cdot(2\cdot 50+1)=50\,050$ reals is about 20% of the 500^2 real size of the full matrix. Thus, if you have data as a matrix then you can hope to compress it with that summation formula by dropping terms with small σ's. The question is whether you lose too much information by only retaining some of the singular values.

To illustrate that you can succeed in retaining at least some aspects of data we will do image compression. Meet Lenna. This top third of a pinup is a standard test image Wikipedia [2012b].

The code we will use is in the `img_squeeze` routine listed at the end of this chapter. The code breaks the picture into three matrices, for the red data, the green data, and the blue data. In that code we have a peek at the eight largest singular values in the red matrix: 93233.7882139, 11193.1186308, 8660.12549856, 7054.38662024, 5916.89456484, 5742.14618029, 4125.16604075, and 3879.15303685. We retain the terms for the largest 10% of the singular values. The code reports that the singular value where we make that cutoff is 606.389279688. It also gives the eight smallest: 0.547288023466, 0.470266870056, 0.1988515109, with five more that are so small they are essentially zero.[1] The large singular values are much larger than the small singular values, even setting aside the ones at the end that are zero except for numerical issues.

We want to see how badly the image degrades for various cutoffs. The code below sets the cutoff at 10%. This image is 512×512 so that will sum the terms associated with the first 51 singular values.

[1] These are four values of $1.024\,723\,452\,83 \times 10^{-11}$ and one of $2.325\,339\,864\,91 \times 10^{-12}$.

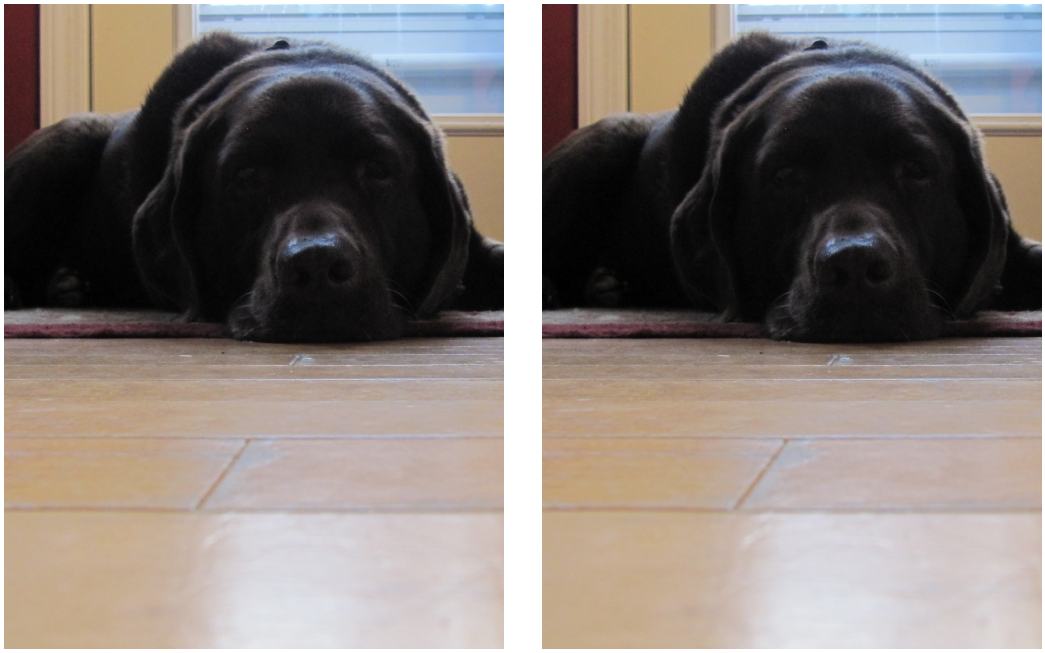
```
sage: runfile("img_squeeze.sage")
sage: img_squeeze("Lenna.png", "Lenna_squeezed.png", 0.10)
```

Below is the squeezed image.

It definitely shows loss. The colors are not as good, the edges are not sharp, and there are a few artifacts, including a horizontal line across the top of Lenna's forehead and another halfway up her hat. But certainly the image is entirely recognizable.

Where the image is $n \times n$, every additional term in the summation adds to the storage and transmission requirements by about $2n$ reals. The original image requires n^2 reals so we want to choose a cutoff value less than 0.50. The above image with a cutoff of 0.10 has the advantage is that it needs only 20% of the storage and transmission requirements of the original image. However, its quality may not be acceptable. That is, selecting a cutoff parameter is an engineering decision where on a fidelity versus resource-consumption continuum you choose a value appropriate for your application.

Experimenting shows that setting the cutoff parameter to 0.20 is enough to make the output image hard to tell from the original. Below is the picture of Suzy from this manual's cover. On the left is the original image and on the right it is squeezed using a parameter value of 0.20.



Source of plot_action.sage

The `plot_circle_action` routine that we call above takes the four entries of the 2×2 matrix and returns a list of graphics. Other parameters are the number of colors, and a flag giving whether to plot a full circle or whether to plot just the top half circle (this is the default).

This routine determines the list of colors and calls a helper `color_circle_list`, given below, which returns a list of graphics. Finally, this routine plots those graphics.

```
def plot_circle_action(a, b, c, d, n = 12, full_circle = False):
    """Show the action of the matrix with entries a, b, c, d on half
    of the unit circle, broken into a number of colors.
     a, b, c, d  reals  Entries are upper left, ur, ll, lr.
     n = 12  positive integer  Number of colors.
     full_circle=False  boolean  Show whole circle, or top half
    """
    colors = rainbow(n)
    G = Graphics()  # holds graph parts until they are to be shown
    for g_part in color_circle_list(a,b,c,d,colors,full_circle):
        G += g_part
    return plot(G)
```

The helper does the heavy lifting. It produces a parametrized curve $(x(t), y(t))$, and uses *Sage*'s `parametric_plot` function to get the resulting graphic.

```
DOT_SIZE = .02
CIRCLE_THICKNESS = 2
```

```
def color_circle_list(a, b, c, d, colors, full_circle=False):
    """Return list of graph instances for the action of a 2x2 matrix on
    half of the unit circle.  That circle is broken into chunks each
    colored a different color.
      a, b, c, d  reals  entries of the matrix ul, ur, ll, lr
      colors  list of rgb tuples; len of this list is how many chunks
      full_circle=False  Show a full circle instead of a half circle.
    """
    r = []
    if full_circle:
        p = 2*pi
    else:
        p = pi
    t = var('t')
    n = len(colors)
    for i in range(n):
        color = colors[i]
        x(t) = a*cos(t)+c*sin(t)
        y(t) = b*cos(t)+d*sin(t)
        g = parametric_plot((x(t), y(t)),
                            (t, p*i/n, p*(i+1)/n),
                            color = color, thickness=CIRCLE_THICKNESS)
        r.append(g)
        r.append(circle((x(p*i/n), y(p*i/n)), DOT_SIZE, color=color))
    if not(full_circle):    # show (x,y)=(-1,0) is omitted
        r.append(circle((x(pi), y(pi)), 2*DOT_SIZE, color='black',
                        fill = 'true'))
        r.append(circle((x(pi), y(pi)), DOT_SIZE, color='white',
                        fill = 'true'))
    return r
```

If this routine is plotting the upper half circle then it adds a small empty circle at the end to show that the image of $(-1,0)$ is not part of the graph.

There are two global values to set graphic values. The variable `CIRCLE_THICKNESS` sets the thickness of the plotted curve, in points (a printer's unit, here 1/72 inch). Similarly `DOT_SIZE` sets the size of the small empty circle.

Source of img_squeeze.sage

We will use the Python Image Library for reading the figure. The function `img_squeeze` takes three arguments, the names of the input and output functions, and a real number between 0 and 1 which determines the percentage of the singular values to include in the sum before the cutoff.

```
# Compress the image
from PIL import Image

def img_squeeze(fn_in, fn_out, percent):
    """Squeeze an image using Singular Value Decomposition.
        fn_in, fn_out  string  name of file
        percent  real in 0..1  Fraction of singular values to use
```

This function first brings the input data to a format where each pixel is a triple (red, green, blue) of integers that range from 0 to 255. It uses those numbers to gradually build three Python arrays `rd`, `gr`, and `bl`, which then initialize the three *Sage* matrices `RD`, `GR`, and `BL`.

```
    img = Image.open(fn_in)
    img = img.convert("RGB")
    rows, cols = img.size
    print "image has",rows,"rows and",cols,"columns"
    cutoff = int(round(percent*min(rows,cols),0))
    # Gather data into three arrays, then give to Sage's matrix()
    rd, gr, bl = [], [], []
    for row in range(rows):
        for a in [rd, gr, bl]:
            a.append([])
        for col in range(cols):
            r, g, b = img.getpixel((int(row), int(col)))
            rd[row].append(r)
            gr[row].append(g)
            bl[row].append(b)
    RD, GR, BL = matrix(RDF, rd), matrix(RDF, gr), matrix(RDF, bl)
```

The next step finds the Singular Value Decomposition of those three. Out of curiosity, we have a peek at the eight largest singular values in the red matrix, the singular value where we make the cutoff, and the eight smallest.

```
    # Get the SVDs
    print "about to get the svd"
    U_RD,Sigma_RD,V_RD = RD.SVD()
    U_GR,Sigma_GR,V_GR = GR.SVD()
    U_BL,Sigma_BL,V_BL = BL.SVD()
    # Have a look
    for i in range(8):
        print "sigma_RD",i,"=",Sigma_RD[i][i]
    print "  :"
    print "sigma_RD",cutoff,"=",Sigma_RD[cutoff][cutoff]
```

Finally, for each matrix we compute the sum $\sigma_1 \cdot \vec{u}_1\vec{v}_1^T + \sigma_2 \cdot \vec{u}_2\vec{v}_2^T + \cdots$ up through the cutoff index.

```
    # Compute sigma_1 u_1 v_1^trans+ ..
    a=[]
    for i in range(rows):
        a.append([])
        for j in range(cols):
            a[i].append(0)
    A_RD, A_GR, A_BL = matrix(RDF, a), matrix(RDF, a), matrix(RDF, a)
    for i in range(cutoff):
        sigma_i = Sigma_RD[i][i]
        u_i = matrix(RDF, U_RD.column(i).transpose())
        v_i = matrix(RDF, V_RD.column(i))
        A_RD = copy(A_RD)+sigma_i*u_i*v_i
        sigma_i = Sigma_GR[i][i]
        u_i = matrix(RDF, U_GR.column(i).transpose())
```

```
        v_i = matrix(RDF, V_GR.column(i))
        A_GR = copy(A_GR)+sigma_i*u_i*v_i
        sigma_i = Sigma_BL[i][i]
        u_i = matrix(RDF, U_BL.column(i).transpose())
        v_i = matrix(RDF, V_BL.column(i))
        A_BL = copy(A_BL)+sigma_i*u_i*v_i
```

The code asks for the transpose of the `U_RD.column(i)`, etc., which may seem to be a mistake. Remember that *Sage* favors rows for vectors, so this is how we make the $\vec{u}$ vector a column, while $\vec{v}$ is already a row. Because of this operation on the vector, the first time you run this code in a *Sage* session you may get a depreciation warning.

To finish, we put the data in the `.png` format and save it to disk.

```
    # Make a new image
    img_squoze = Image.new("RGB", img.size)
    for row in range(rows):
        print "transferring over row=", row
        for col in range(cols):
            p = (int(A_RD[row][col]),
                 int(A_GR[row][col]),
                 int(A_BL[row][col]))
            img_squoze.putpixel((int(row), int(col)), p)
    img_squoze.save(fn_out)
```

This part of the routine takes a long time, in part because the code is intended to be easy to read rather than fast. Consequently, in the source there are some status lines to convince a user that it is still working; those busy-work lines are left out of some of the earlier output listings.

Geometry of Linear Maps

Sage can illustrate the geometric effect of linear transformations. In this chapter we will picture their effect on the unit square. The graphs here show transformations of the plane $\mathbb{R}^2$ because those fit on the paper but the principles extend to higher-dimensional spaces.

Lines map to lines

The prior chapter points out that the condition $h(r\vec{v}) = r \cdot h(\vec{v})$ in the definition of linear map means that linear maps send lines through the origin to lines through the origin. We can extend that to show that under a linear map, the image of any line at all in the domain is a line in the range. Fix a domain space $\mathbb{R}^d$ and codomain space $\mathbb{R}^c$, along with the linear map h between them. Get a line in the domain by fixing a vector of slopes $\vec{m} \in \mathbb{R}^d$ and a vector of offsets from the origin $\vec{b} \in \mathbb{R}^d$, so $\ell = \{\vec{v} = \vec{m} \cdot s + \vec{b} \mid s \in \mathbb{R}\}$. The image of ℓ is this set.

$$h(\ell) = \{h(\vec{m} \cdot s + \vec{b}) \mid s \in \mathbb{R}\} = \{h(\vec{m}) \cdot s + h(\vec{b}) \mid s \in \mathbb{R}\}$$

This is a line in the codomain with the vector of slopes $h(\vec{m})$ and the vector of offsets $h(\vec{b})$.

For example, consider the transformation $t\colon \mathbb{R}^2 \to \mathbb{R}^2$ that rotates vectors counterclockwise by $\pi/6$ radians.[1]

$$\mathrm{Rep}_{E_2,E_2}(t) = \begin{pmatrix} \cos(\pi/6) & \sin(\pi/6) \\ -\sin(\pi/6) & \cos(\pi/6) \end{pmatrix} = \begin{pmatrix} \sqrt{3}/2 & 1/2 \\ -1/2 & \sqrt{3}/2 \end{pmatrix}$$

This is the line $y = 3x + 2$.

$$\ell = \{\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 3 \\ 1 \end{pmatrix} \cdot s + \begin{pmatrix} 2 \\ 0 \end{pmatrix} \mid s \in \mathbb{R}\}$$

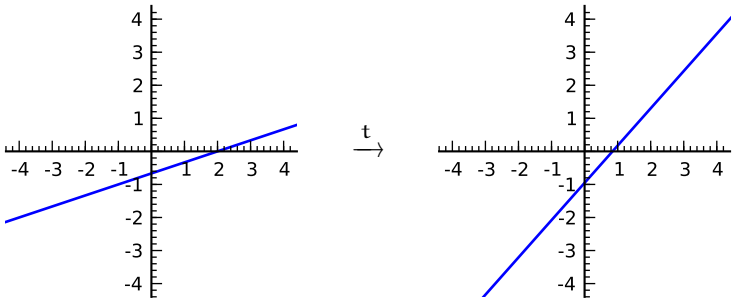
Under the action of t that line becomes this set.

$$t(\ell) = \{\begin{pmatrix} x \\ y \end{pmatrix} = \tfrac{1}{2}\begin{pmatrix} 3\sqrt{3}-1 \\ 3+\sqrt{3} \end{pmatrix} \cdot s + \begin{pmatrix} \sqrt{3} \\ 1 \end{pmatrix} \mid s \in \mathbb{R}\}$$

```
sage: s = var('s')
sage: ell = parametric_plot((3*s+2,1*s), (s, -10, 10))
sage: ell.set_axes_range(-4, 4, -4, 4)
sage: ell.save("graphics/geo000a.pdf")
```

[1] Recall that *Sage* prefers to do matrix-vector multiplication as $\vec{v}M$ (here $\vec{v}$ is a row vector). Translate to the book's convention by transposing, $M^T\vec{v}^T$. See the discussion on page 43.

```
sage: t_x(s) = ((3*sqrt(3)-1)/2)*s+sqrt(3)
sage: t_y(s) = ((3+sqrt(3))/2)*s+1
sage: t_ell = parametric_plot((t_x(s), t_y(s)), (s, -10, 10))
sage: t_ell.set_axes_range(-4, 4, -4, 4)
sage: t_ell.save("graphics/geo000b.pdf")
```

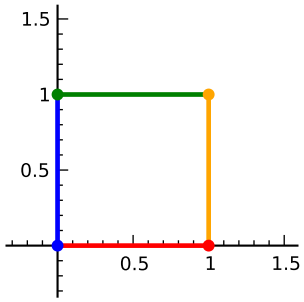


(The limits on the parameter s of 10 and -10 are arbitrary, just chosen to be large enough that the line segment covers the entire domain and codomain intervals shown, from -4 to 4.) One instance is that the vector that ends at $(2,0)$ is rotated to the vector that ends at $(\sqrt{3},1)$.

The unit square

We can show the effect of plane transformations $t\colon \mathbb{R}^2 \to \mathbb{R}^2$ by applying them to the unit square.

```
sage: runfile("plot_action.sage")
sage: p = plot_square_action(1,0,0,1)  # identity matrix
sage: p.set_axes_range(-0.25, 1.5, -0.25, 1.5)
sage: p.save("graphics/geo100.pdf")
```



The `plot_square_action(a,b,c,d)` routine plots the result of applying to a unit square the transformation represented with respect to the standard basis by the matrix with entries a, b, c, and d.

$$(x\ \ y)\begin{pmatrix} a & b \\ c & d \end{pmatrix} = (ax+cy\ \ \ bx+dy)$$

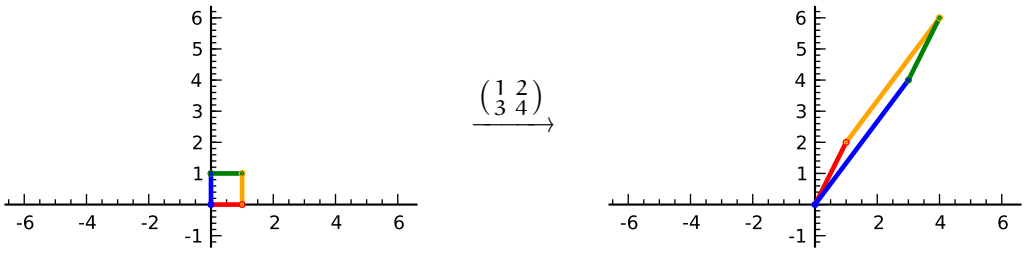
The routine's source is at the end of this chapter but the observation that linear maps send lines to lines makes it easy: the matrix does this to the four corners of the square

$$\begin{pmatrix}0\\0\end{pmatrix} \overset{t}{\longmapsto} \begin{pmatrix}0\\0\end{pmatrix} \quad \begin{pmatrix}1\\0\end{pmatrix} \overset{t}{\longmapsto} \begin{pmatrix}a\\b\end{pmatrix} \quad \begin{pmatrix}1\\1\end{pmatrix} \overset{t}{\longmapsto} \begin{pmatrix}a+c\\b+d\end{pmatrix} \quad \begin{pmatrix}0\\1\end{pmatrix} \overset{t}{\longmapsto} \begin{pmatrix}c\\d\end{pmatrix}$$

and the square's sides between those corners map to the line segments between those image points — that is, linear transformations map squares to parallelograms. So, the routine just plots the four line segments connecting those corners. In the *Sage* session above, that routine is given the identity matrix, so it plots the unit square unchanged.

That routine gives these pictures of the effect of the generic matrix.

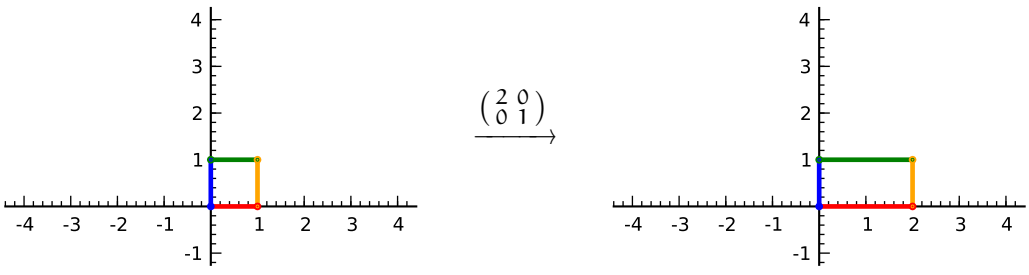
```
sage: runfile("plot_action.sage")
sage: q = plot_square_action(1,0,0,1)
sage: q.set_axes_range(-6, 6, -1, 6)
sage: q.save("graphics/geo101a.pdf")
sage: p = plot_square_action(1,2,3,4)
sage: p.set_axes_range(-6, 6, -1, 6)
sage: p.save("graphics/geo101b.pdf")
```



The colors show another effect of transformations, beyond shape-changing while line-preserving. Take the colors in their natural order of red, orange, green, and blue. Then the domain square has a counterclockwise orientation while the codomain's figure is clockwise, so the colors illustrate whether the transformation preserves or reverses orientation.

We can develop an understanding of complex behavior by building on an understanding of simple behavior. This transformation doubles the x components of all vectors.[1]

```
sage: p = plot_square_action(2,0,0,1)
```



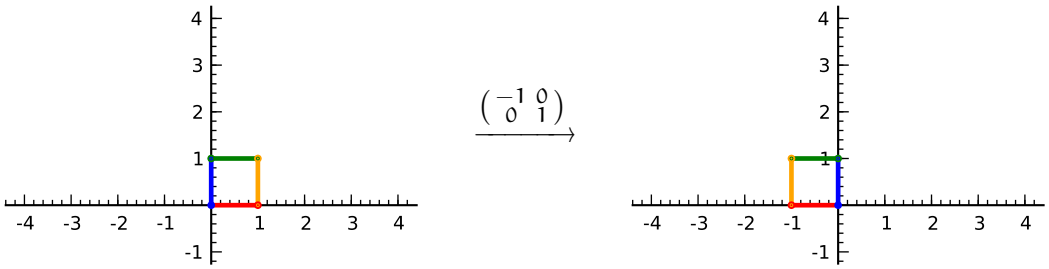
(*)

[1] In the rest of this section we'll omit the fiddly lines that load the script file, set the axis ranges, etc.

Linear maps send the zero vector to the zero vector, and the input square is anchored at the origin, so the output shape is also anchored at the origin. But it has been stretched horizontally — it has the same orientation as the starting square, but twice the area.

That example illustrates that the behavior associated with diagonal matrices is simple. For instance, tripling the x coordinate gives you a similar shape with three times the area of the starting one. What about if you take -1 times the x-coordinate?

```
sage: p = plot_square_action(-1,0,0,1)
```



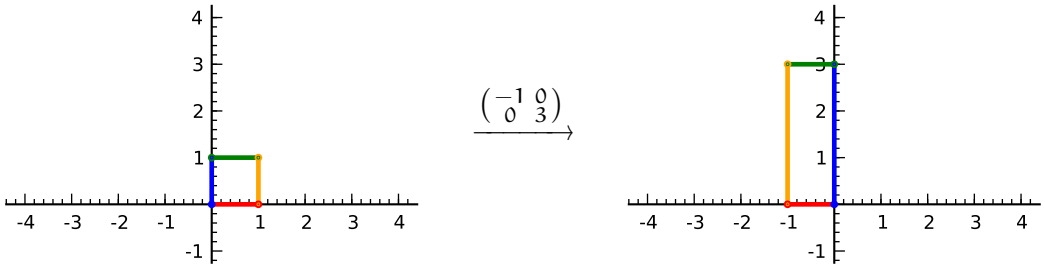
(**)

It changes the orientation.

We say that the above shape has an *oriented area* of -1. The motivation for taking the area with a sign is: imagine starting with the right-hand figure from the example before this one and slide the orange side in from the right, from 2 to 1, to 0 and then to -1. The area falls from 2 to 1, to 0, and so we naturally to assign the figure above an area measure of -1. The prefix 'oriented' is just there to distinguish this idea from the grade school meaning of area. That meaning of area is the absolute value of the oriented area.

The next transformation combines action in two axes, tripling the y components and multiplying x components by -1.

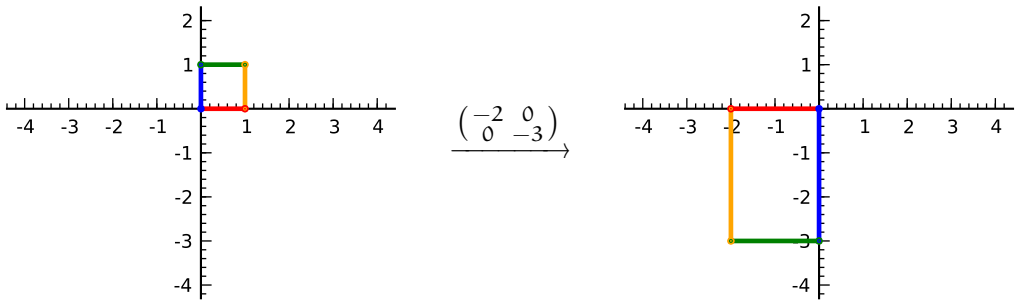
```
sage: p = plot_square_action(-1,0,0,3)
```



The colors show that this transformation also changes the orientation, so the new shape has an oriented area of -3.

What if we change the orientation twice?

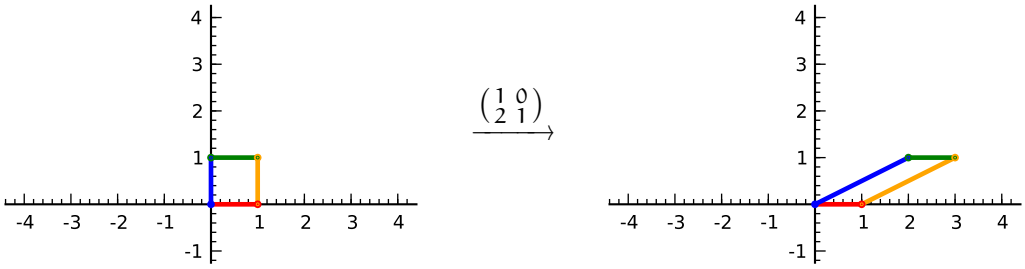
```
sage: p = plot_square_action(-2,0,0,-3)
```



The colors are the same as the original shape's counterclockwise red, orange, green, and then blue. Thus the new shape has an oriented area of 6.

We next show the effect of putting in off-diagonal entries.

```
sage: p = plot_square_action(1,0,2,1)
```



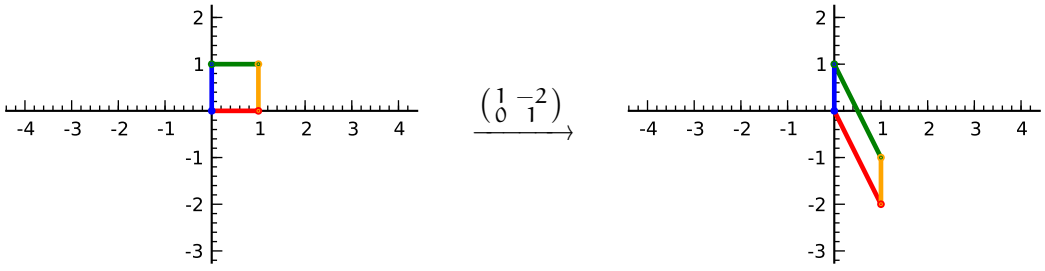
This transformation is a *skew*[1] (or *shear*). The line segment sides of the original square map to line segments, but the sides are not at right angles. The action

$$\begin{pmatrix} x \\ y \end{pmatrix} \mapsto \begin{pmatrix} x+2y \\ y \end{pmatrix}$$

means that a starting vector with a y component of 1 gets shifted right by 2 while a starting vector with a y component of 2 is shifted right by 4, so vectors are shifted depending on how far they are above or below the x-axis. This transformation preserves orientation and the output shape has a base of 1 with a height of 1 so its oriented area is 1.

Putting a nonzero value in the other off-diagonal entry of the matrix, the upper right, has the same effect except that it skews parallel to the y-axis.

```
sage: p = plot_square_action(1,-2,0,1)
```



[1] The word skew means "to distort from a symmetrical form."

The action

$$\begin{pmatrix} x \\ y \end{pmatrix} \mapsto \begin{pmatrix} x \\ -2x+y \end{pmatrix}$$

means that vectors are shifted depending on how far they are from the x axis. For instance, a vector with an x component of 1 is shifted by -2. The oriented area of the output shape is 1.

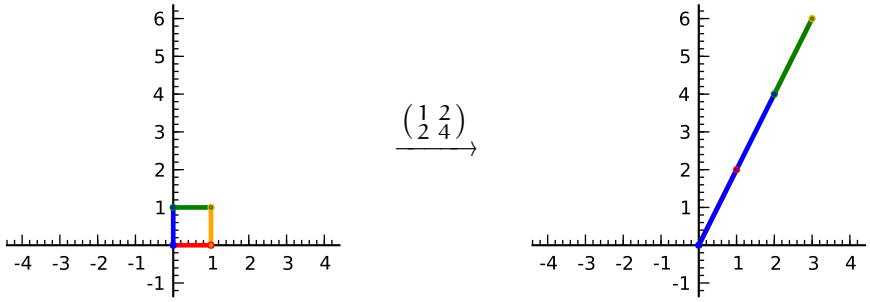
Determinants

The book geometrically interprets the conditions in the definition of a determinant function. It shows that, in going from the before picture to the after, these transformation matrices change the oriented area of the input region by a factor that is the determinant of the matrix. In $(*)$ the matrix has determinant 2 and it doubles the oriented area. In $(**)$ the matrix multiplies the oriented area by -1.

One advantage of starting these before/after pictures with a unit square is that then the output shape has oriented area equal to the determinant of the matrix.

For instance, this matrix is singular, so it has determinant 0.

```
sage: p = plot_square_action(1,2,2,4)
```



The output shape has zero content.

Turing's factorization PA=LDU

We will now see how the action of any matrix can be decomposed into the actions shown above. This will give us a complete geometric description of any linear map, that is, you can understand the effect of any transformation by breaking down into a sequence of steps that are simple.

Recall that you can do the row operations of Gauss's Method with matrix multiplication. For instance, multiplication from the left by this matrix has the effect of the row operation $2\rho_1 + \rho_2$.

$$\begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 3 & 1 & 4 \\ -6 & 1 & -8 \\ 0 & -3 & 2 \end{pmatrix} = \begin{pmatrix} 3 & 1 & 4 \\ 0 & 3 & 0 \\ 0 & -3 & 2 \end{pmatrix}$$

In general, as described in the book, the *elementary reduction matrices* come in three types $M_i(k)$, $P_{i,j}$, and $C_{i,j}(k)$, and arise from applying a row operation to an identity matrix.

$$I \xrightarrow{k\rho_i} M_i(k) \text{ for } k \neq 0 \qquad I \xrightarrow{\rho_i \leftrightarrow \rho_j} P_{i,j} \text{ for } i \neq j \qquad I \xrightarrow{k\rho_i + \rho_j} C_{i,j}(k) \text{ for } i \neq j$$

For a matrix H you can do row scaling $k\rho_i$ with $M_i(k)H$, you can swap rows $\rho_i \leftrightarrow \rho_j$ with $P_{i,j}H$, and you can add a multiple of one row to another $k\rho_i + \rho_j$ with $C_{i,j}(k)H$. The prior paragraph used the 3×3 matrix $C_{1,2}(2)$. (We are focused on transformations so we will assume all of these matrices are same-sized and square.)

You can continue the Gauss's Method started in the equation above and use $C_{2,3}(-1)$ to perform $-\rho_2 + \rho_3$, producing echelon form.

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 3 & 1 & 4 \\ -6 & 1 & -8 \\ 0 & -3 & 2 \end{pmatrix} = \begin{pmatrix} 3 & 1 & 4 \\ 0 & 3 & 0 \\ 0 & 0 & 2 \end{pmatrix} \qquad (*)$$

As in this example, matrix multiplication by these elementary matrices suffices to do Gauss's Method and produce echelon form.

Observe further that if the starting matrix is such that you don't need any row swapping then you can stick with the operations $k\rho_i + \rho_j$ where $j > i$ (that is, where a row operates on a row below it). The elementary matrices that perform those operations are *lower triangular* since all of their nonzero entries are in the lower left. (Matrices with all of their nonzero entries in the upper right are *upper triangular*.) Thus your factorization so far is into a product of lower triangular elementary matrices and an echelon form matrix.

You can go further. Use a diagonal matrix to make the leading entries of the nonzero rows of the echelon form matrix into 1's. Here is that additional step performed on the above equation.

$$\begin{pmatrix} 1/3 & 0 & 0 \\ 0 & 1/3 & 0 \\ 0 & 0 & 1/2 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 3 & 1 & 4 \\ -6 & 1 & -8 \\ 0 & -3 & 2 \end{pmatrix} = \begin{pmatrix} 1 & 1/3 & 4/3 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \qquad (*)$$

You can use multiplication by elementary matrices to go all the way to a block partial identity matrix. The idea is to use column operations. Here is right-multiplication on the right-hand side of $(*)$ to add $-1/3$ times the first column to the second column.

$$\begin{pmatrix} 1 & 1/3 & 4/3 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & -1/3 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 4 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Then adding $-4/3$ times the first column to the third column leave an identity matrix.

$$\begin{pmatrix} 3 & 1 & 4 \\ 0 & 3 & 0 \\ 0 & 0 & 2 \end{pmatrix} \begin{pmatrix} 1 & 0 & -4/3 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \qquad (**)$$

Thus, if you start with a matrix A that does not require any row swaps then you get this matrix equation.

$$L_1 L_2 \cdots L_k \cdot A \cdot U_1 U_2 \cdot U_r = D$$

Here D is a partial identity matrix, the L_i are lower-triangular row combination matrices, and the U_j are upper-triangular column combination matrices.

All of the row operations can be undone (for instance, $2\rho_1+\rho_2$ is undone with $-2\rho_1+\rho_2$), Thus each of those lower triangular matrices has an inverse. Likewise, each upper-triangular matrix has an inverse. Therefore, if you don't need any swaps in a Gauss-Jordan reduction of a matrix A then you have this factorization.

$$A = L_k^{-1} \cdots L_1^{-1} \cdot D \cdot U_r^{-1} \cdots U_1^{-1}$$

To fix the swap issue you can pre-swap: before factoring the starting matrix, first swap its rows with a permutation matrix P.

$$P \cdot A = L_k^{-1} \cdots L_1^{-1} \cdot D \cdot U_r^{-1} \cdots U_1^{-1} \qquad (***)$$

A product of lower triangular matrices is lower-triangular, and a product of upper triangular matrices is upper-triangular so you can combine all the L's and all the U's, and the formula $(***)$ is often known as $PA = LDU$.

We illustrate with the generic 2×2 transformation of $\mathbb{R}^2$ represented with respect to the standard basis in this way.

$$T = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

Gauss's Method is straightforward.

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \xrightarrow{-3\rho_1+\rho_2} \begin{pmatrix} 1 & 2 \\ 0 & -2 \end{pmatrix} \xrightarrow{-(1/2)\rho_2} \begin{pmatrix} 1 & 2 \\ 0 & 1 \end{pmatrix} \xrightarrow{-2\chi_1+\chi_2} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

(We use χ_i for the columns.) This is the associated factorization.

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 3 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & -2 \end{pmatrix} \begin{pmatrix} 1 & 2 \\ 0 & 1 \end{pmatrix}$$

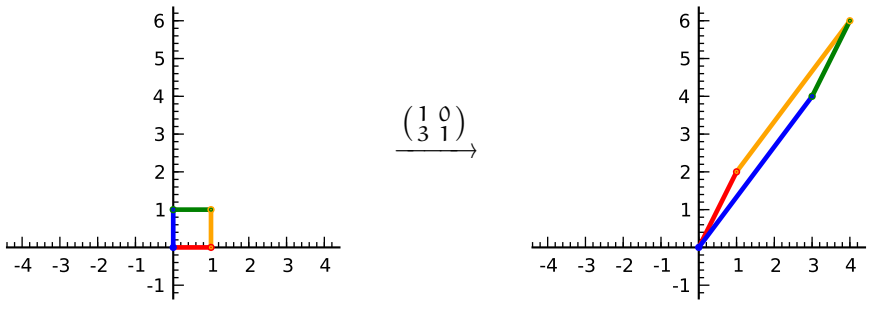
The product checks out.

```
sage: L = matrix(QQ, [[1, 0], [3, 1]])
sage: D = matrix(QQ, [[1, 0], [0, -2]])
sage: U = matrix(QQ, [[1, 2], [0, 1]])
sage: L*D*U
[1 2]
[3 4]
```

We got into this to understand the geometric effect of the generic transformation.

$$(x \;\; y) \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

```
sage: p = plot_square_action(1,2,3,4)
```



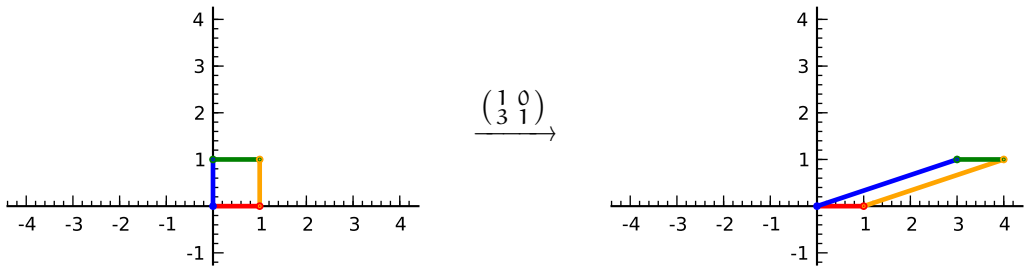
Expand it using the above LDU factorization.

$$(x\ y)\begin{pmatrix}1&0\\3&1\end{pmatrix}\begin{pmatrix}1&0\\0&-2\end{pmatrix}\begin{pmatrix}1&2\\0&1\end{pmatrix}$$

We will look twice at the effects these three. First we will see the effect of each of L, D, and U separately. Then we will look at the cumulative effect: that of L, then of LD, and finally LDU.

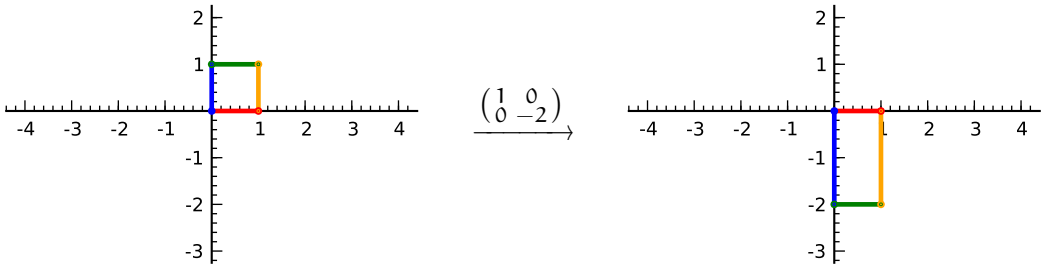
The first matrix is a skew parallel to the x-axis.

```
sage: p = plot_square_action(1,0,3,1)
```



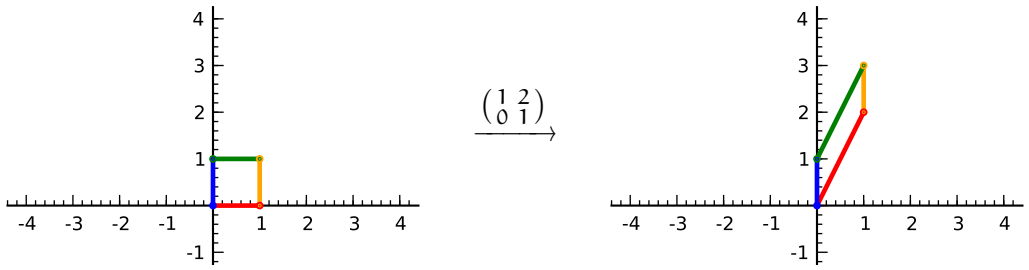
The second matrix rescales and changes the orientation.

```
sage: p = plot_square_action(1,0,0,-2)
```



Finally, the third matrix is a skew parallel to the y-axis.

```
sage: p = plot_square_action(1,2,0,1)
```



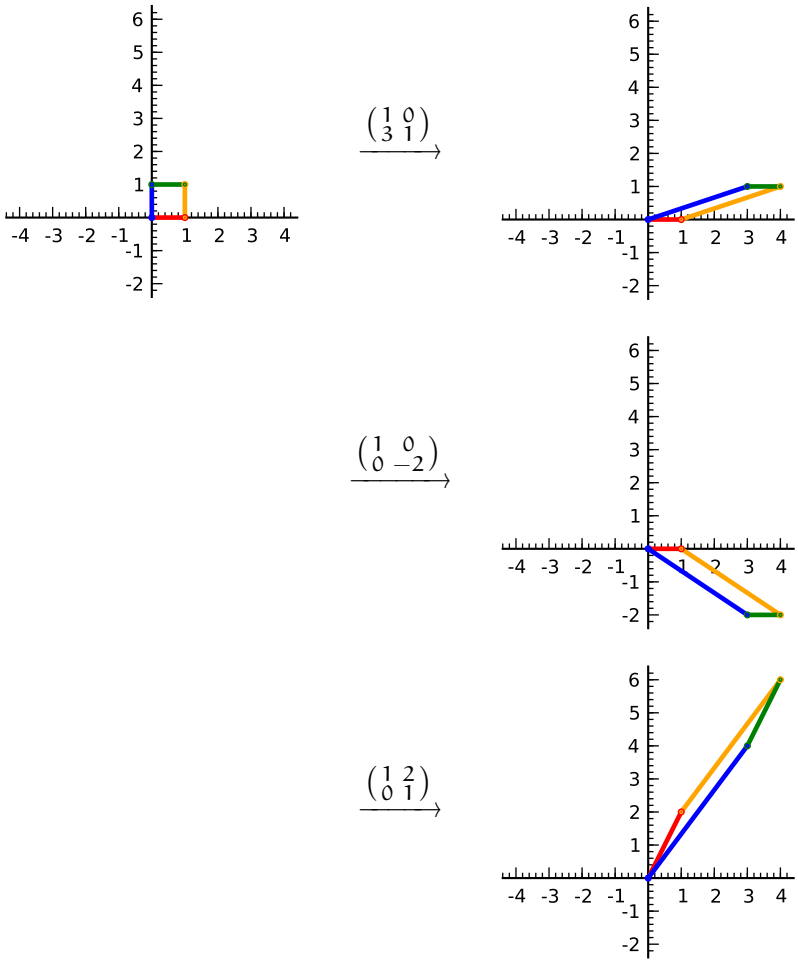
The lower-triangular and upper-triangular matrices do not change orientation. Any orientation changing in LDU happens via diagonal entries that are negative. (*Comment.* However, if a

matrix requires row swaps $PA = LDU$ then the situation is more subtle. Each row swap toggles the orientation, from counterclockwise to clockwise or from clockwise to counterclockwise. Thus if the permutation matrix requires an odd number of swaps then it changes the orientation but with an even number of swaps it leaves the orientation the same.)

Now we look at the cumulative effect: the actions of L, then LD, then LDU. For example, here is the cumulative effect of the maps on the unit square's upper right corner.

$$\begin{pmatrix}1\\1\end{pmatrix} \overset{\text{L}}{\longmapsto} \begin{pmatrix}4\\1\end{pmatrix} \overset{\text{D}}{\longmapsto} \begin{pmatrix}4\\-2\end{pmatrix} \overset{\text{U}}{\longmapsto} \begin{pmatrix}4\\6\end{pmatrix}$$

```
sage: L = matrix(QQ, [[1, 0], [3, 1]])
sage: D = matrix(QQ, [[1, 0], [0, -2]])
sage: U = matrix(QQ, [[1, 2], [0, 1]])
sage: LD = L*D
sage: LDU = LD*U
sage: p = plot_square_action(1,0,0,1)
sage: p = plot_square_action(L[0][0], L[0][1], L[1][0], L[1][1])
sage: p = plot_square_action(LD[0][0], LD[0][1], LD[1][0], LD[1][1])
sage: p = plot_square_action(LDU[0][0], LDU[0][1], LDU[1][0],LDU[1][1])
```



Source of plot_action.sage

The `plot_square_action` routine takes the four entries of the 2×2 matrix and returns a list of graphics. Most of the heavy lifting here is done by the helper `color_square_list`, given below.

```
def plot_square_action(a, b, c, d):
    """Show the action of the matrix with entries a, b, c, d on half
    of the unit circle, as the circle and the output curve, broken into
    colors.
     a, b, c, d  reals  Entries are upper left, ur, ll, lr.
    """
    colors = ['red', 'orange', 'green', 'blue']
    G = Graphics()         # hold graph parts until they are to be shown
    for g_part in color_square_list(a,b,c,d,colors):
        G += g_part
    p = plot(G)
    return p
```

There are two technical points about the helper routine that could do with explanation. First is `ZORDER`, which determines the order in which *Sage* plots things, and here we want the unit square to be plotted after the axes (so its colors will be visible). The second is that the way line segments butt against each other is ugly so we cover the butt with a dot.

```
SQUARE_THICKNESS = 1.75  # How thick to draw the curves
ZORDER = 5    # Draw the graph over the axes
def color_square_list(a, b, c, d, colors):
    """Return list of graph instances for the action of a 2x2 matrix
    on a unit square.  That square is broken into sides, each colored a
    different color.
      a, b, c, d  reals  entries of the matrix
      colors  list of rgb tuples; len of this list is at least four
    """
    r = []
    t = var('t')
    # Four sides, ccw around square from origin
    r.append(parametric_plot((a*t, b*t), (t, 0, 1),
                             color = colors[0], zorder=ZORDER,
                             thickness = SQUARE_THICKNESS))
    r.append(parametric_plot((a+c*t, b+d*t), (t, 0, 1),
                             color = colors[1], zorder=ZORDER,
                             thickness = SQUARE_THICKNESS))
    r.append(parametric_plot((a*(1-t)+c, b*(1-t)+d), (t, 0, 1),
                              color = colors[2], zorder=ZORDER,
                              thickness = SQUARE_THICKNESS))
    r.append(parametric_plot((c*(1-t), d*(1-t)), (t, 0, 1),
                             color = colors[3], zorder=ZORDER,
                             thickness = SQUARE_THICKNESS))
    # Dots make a cleaner join between edges
    r.append(circle((a, b), DOT_SIZE,
                    color = colors[0], zorder = 2*ZORDER,
                    thickness = SQUARE_THICKNESS*1.25, fill =  True))
    r.append(circle((a+c, b+d), DOT_SIZE,
                    color = colors[1], zorder = 2*ZORDER+1,
```

```
                    thickness = SQUARE_THICKNESS*1.25, fill =  True))
    r.append(circle((c, d), DOT_SIZE,
                    color = colors[2], zorder = ZORDER+1,
                    thickness = SQUARE_THICKNESS*1.25, fill =  True))
    r.append(circle((0, 0), DOT_SIZE,
                    color = colors[3], zorder = ZORDER+1,
                    thickness = SQUARE_THICKNESS*1.25, fill =  True))
    return r
```

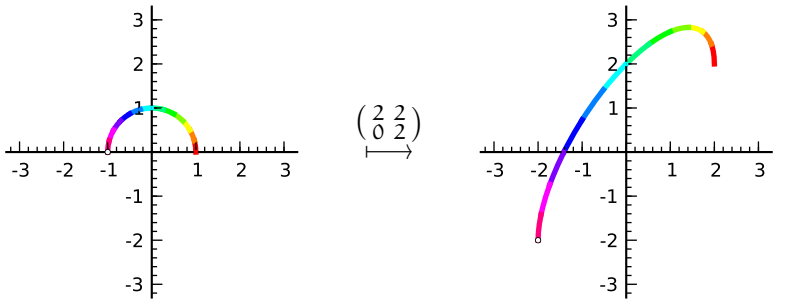
Eigenvalues

In Chapter 6 on Singular Value Decomposition we studied how transformations resize vectors; the maximum resizing factors are the singular values. In this chapter we consider another geometric effect.

Turning

Consider again the geometry of the action of a skew map on the unit circle.[1]

```
sage: runfile("plot_action.sage")
sage: q = plot_circle_action(1,0,0,1)
sage: q.set_axes_range(-3, 3, -3, 3)
sage: q.save("graphics/eigen000a.pdf")
sage: p = plot_circle_action(2,2,0,2)
sage: p.set_axes_range(-3, 3, -3, 3)
sage: p.save("graphics/eigen000b.pdf")
```

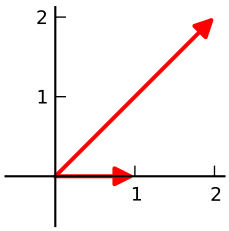


$(*)$

On the left plot, going counterclockwise, the curve begins with red at $(x, y) = (1, 0)$. On the right plot it begins with red at $(1, 2)$. Here are the before and after vectors for that red point.

```
sage: p = plot_before_after_action(2,2,0,2, [(1,0)], ['red'])
```

[1] Recall that *Sage* prefers to do matrix-vector multiplication as $\vec{v}M$ (here $\vec{v}$ is a row vector). Translate to the book's convention by transposing, $M^{\mathsf{T}}\vec{v}^{\mathsf{T}}$. See the discussion on page 43.



The matrix's action on the red vector is to both resize and rotate.

```
sage: v = vector(RR, [1,0])
sage: M = matrix(RR, [[2, 2], [0, 2]])
sage: w = v*M
sage: w.norm(), v.norm()
(2.82842712474619, 1.00000000000000)
sage: angle = arccos(w*v/(w.norm()*v.norm()))
sage: angle
0.785398163397448
```

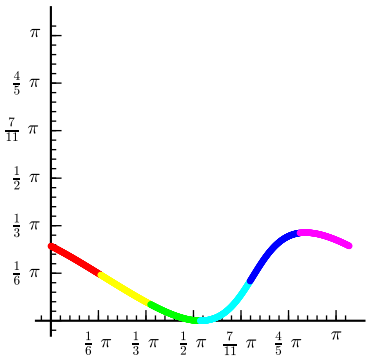
The map resizes and rotates the violet vector at the other end of the half circle, at $(-1, 0)$, by the same amount as it does the red vector ending at $(1, 0)$. This is because a linear map has the same action on all vectors that lie on the same line through the origin.

But other vectors may be resized and rotated by the other amounts. Here is the effect of the transformation on the point one sixth of the way around the half-circle.

```
sage: v = vector(RR, [cos(pi/6),sin(pi/6)])
sage: M = matrix(RR, [[2, 2], [0, 2]])
sage: w = v*M
sage: w.norm(), v.norm()
(3.23482636553150, 1.00000000000000)
sage: angle = arccos(w*v/(w.norm()*v.norm()))
sage: angle
0.482170608511459
```

Sage will compute for us the rotations of the vectors. At the end of this chapter is the source for a routine `plot_color_angles` that picks many vectors in the half circle, applies the transformation to each, computes the angle by which that vector is rotated, and draws a scatter plot. That scatter plot shows the points using the colors of the input vectors. Because there are many points, the result looks like a smooth curve.

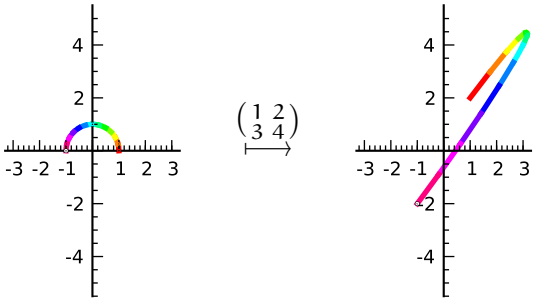
```
sage: p = plot_color_angles(2,2,0,2)
sage: p.set_axes_range(0,pi,0,pi)
sage: p.save("graphics/eigen003.pdf", figsize=3, tick_formatter=[pi,pi])
```



The most interesting point on this graph is where the output angle is 0, at the input angle of $\pi/2$. This is a input vector that the transformation does not turn at all. It is resized but not rotated. This appears in this chapter's first graphic, the one labeled (*) on page 77, as the green input vector lying on the y-axis that is associated with the green output that also lies on the y-axis.

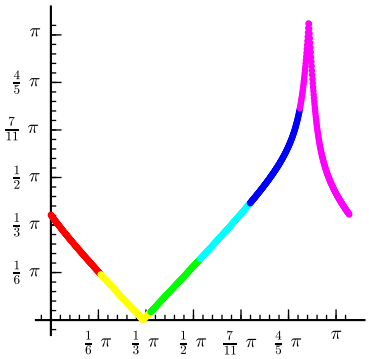
Generic matrix We can do the same analysis for our usual generic 2×2 matrix. Here is its action on the unit circle.

```
sage: p = plot_circle_action(1,2,3,4)
```



This graphs the angle between the each before arrow and its associated after arrows.

```
sage: p = plot_color_angles(1,2,3,4)
sage: p.set_axes_range(0,pi,0,pi)
sage: p.save("graphics/eigen101.pdf", tick_formatter=[pi,pi])
```



(**)

This graph has two interesting points, where $y = 0$ and where $y = \pi$. In both places the vector is not turned at all, only resized. In the second case the input vector is on the same line through the origin as the output, but it gets rescaled by a negative number.

A vectors that is not turned by a transformation t but instead is purely resized is an *eigenvector* for t and the amount by which it is resized is the *eigenvalue* for that vector.

For an eigenvalue λ, the set of vectors associated with it is an *eigenspace*.

```
sage: M = matrix(RDF, [[1, 2], [3, 4]])
sage: evs = M.eigenvectors_left()
sage: evs
[(-0.372281323269, [(-0.909376709132, 0.415973557919)], 1),
(5.37228132327, [(-0.565767464969, -0.824564840132)], 1)]
sage: evs[0]
(-0.372281323269, [(-0.909376709132, 0.415973557919)], 1)
sage: evs[1]
(5.37228132327, [(-0.565767464969, -0.824564840132)], 1)
```

A matrix has the same eigenvalues whether we are taking multiplication by vectors to come from the left $\vec{v}M$ or from the right $M\vec{v}$. But the eigenvectors will be different. The *Sage* operation `eigenvectors_left` covers the $\vec{v}M$ case and naturally `eigenvectors_right` covers the other.

When computing the eigenvectors and eigenvalues, *Sage* gives a list with two elements. The first element, `evs[0]`, says that the set of vectors with a basis consisting of the single unit vector with endpoint approximately $(-0.91, 0.42)$ is an eigenspace associated with the eigenvalue approximately equal to -0.37 (we ignore the trailing 1 here). The second element, `evs[1]`, says that *Sage* has found a second eigenspace whose a basis consisting of the single vector with endpoint around $(-0.57, -0.82)$ that is associated with the eigenvalue that is about 5.37.

Sage will tell us which of those vectors is which on the graph labeled (**).

```
sage: M = matrix(RDF, [[1, 2], [3, 4]])
sage: evs = M.eigenvectors_left()
sage: v = vector(RDF, evs[0][1][0])
sage: angle_v = atan2(v[1], v[0])
sage: n(angle_v/pi)
0.863440883138597
```

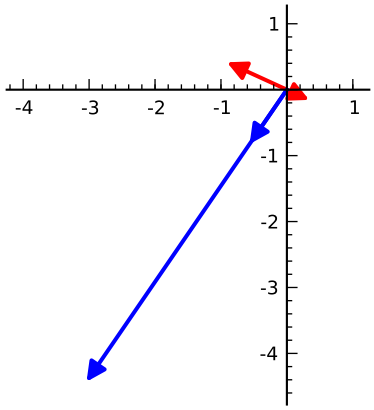
(Remember that the `n()` function gives the numerical value of the argument.) So the eigenspace listed first is the one associated with the right-hand of the two interesting points in (**), the vector that is turned by an angle of π. That dovetails with the observation that the eigenvalue is a negative number because both say that the transformation's action in passing from the domain to the codomain is to turn the vector around.

Sage can draw before and after pictures for the two eigenvectors.

```
sage: p1 = plot_before_after_action(1,2,3,4, [evs[0][1][0]], ['red'])
sage: p2 = plot_before_after_action(1,2,3,4, [evs[1][1][0]], ['blue'])
sage: p = p1+p2
sage: p.set_axes_range(-4, 1, -4.5, 1)
sage: p.save("graphics/eigen102.pdf", figsize=3.25)
```

This picture has two pair of before and after vectors. One pair is in blue and one is in red. Each before vector is a unit vector so it is easy to pick it from among the two. Again we see that the after vector is scaled from the before vector, in the blue case by the positive factor $\lambda_2 \approx 5.37$ and

in the red case by the negative factor $\lambda_1 \approx -0.37$.



Matrix polynomials

Sage will find characteristic and minimal polynomials of a matrix.

```
sage: M =  matrix(RDF, [[1, 2], [3, 4]])
sage: poly = M.charpoly()
sage: poly.factor()
(x - 5.37228132327) * (x + 0.372281323269)
sage: poly.roots()
[(-0.372281323269, 1), (5.37228132327, 1)]
```

The characteristic polynomial and minimal polynomial differ only when the characteristic polynomial has repeated roots.

```
sage: M =  matrix(RDF, [[2, 2, 3], [0, 4, -1], [0, 0, 2]])
sage: M.charpoly()
x^3 - 8.0*x^2 + 20.0*x - 16.0
sage: M.minpoly()
x^2 - 6.0*x + 8.0
sage: M.charpoly().factor()
(x - 4.0) * (x^2 - 4.0*x + 4.0)
sage: M.minpoly().factor()
(x - 4.0) * (x - 2.0)
```

Note here that *Sage* has trouble there telling whether 2 is a repeated root. If we do an exact calculation using rational numbers then we get the right answer.

```
sage: M =  matrix(QQ, [[2, 2, 3], [0, 4, -1], [0, 0, 2]])
sage: M.charpoly()
x^3 - 8*x^2 + 20*x - 16
sage: M.minpoly()
x^3 - 8*x^2 + 20*x - 16
sage: M.charpoly().factor()
(x - 4) * (x - 2)^2
sage: M.minpoly().factor()
(x - 4) * (x - 2)^2
```

Diagonalization and Jordan form

Sage will tell us if two matrices are similar.

```
sage: S =  matrix(QQ, [[2, -3], [1, -1]])
sage: T =  matrix(QQ, [[0, -1], [1,  1]])
sage: S.is_similar(T)
True
sage: U =  matrix(QQ, [[1, 2], [3,  4]])
sage: S.is_similar(U)
False
```

We can determine if a matrix is diagonalizable.

```
sage: M =  matrix(QQ, [[4, -2], [1, 1]])
sage: M.is_diagonalizable()
True
```

To diagonalize the matrix put it in Jordan form.

```
sage: M =  matrix(QQ, [[2, -2, 2], [0, 1, 1], [-4, 8, 3]])
sage: M.jordan_form()
[3|0|0]
[-+-+-]
[0|2|0]
[-+-+-]
[0|0|1]
```

Note the `-+-+-` lines that break the matrix into its component blocks.

Sage will even give you an appropriate transformation matrix.

```
sage: JF, T = M.jordan_form(transformation=True)
sage: JF
[3|0|0]
[-+-+-]
[0|2|0]
[-+-+-]
[0|0|1]
sage: T
[  1   1   1]
[1/2 4/9 1/2]
[  1 4/9   0]
sage: T*JF*T^(-1)
[ 2 -2  2]
[ 0  1  1]
[-4  8  3]
```

We can find the Jordan form of any matrix.

```
sage: M =  matrix(QQ, [[2, -1], [1, 4]])
sage: M.jordan_form()
[3 1]
[0 3]
```

Source of plot_color_angles

This routine gathers graphic instances and returns the plot of that list.

```
def plot_color_angles(a, b, c, d, num_points=1000):
    """Show the action of the matrix with entries a, b, c, d on half
    of the unit circle, broken into a number of colors.
     a, b, c, d  reals  Entries are upper left, ur, ll, lr.
     num_points=1000  Number of points along half circle to plot
    """
    # colors = rainbow(num_points*1.5)[:num_points]   # just a hack
    colors = rainbow(6)
    G = Graphics()  # holds graph parts until they are to be shown
    for g_part in color_angles_list(a,b,c,d,num_points,colors):
        G += g_part
    return plot(G)
```

Source of color_angles_list

This finds the angles for the effect of the transformation on the vectors, and draws a scatter plot of those points. It draws them in the color of the input vector.

```
MARKERSIZE = 2
TICKS = ([0,pi/4,pi/2,3*pi/4,pi], [0,pi/2,pi])
def color_angles_list(a, b, c, d, num_pts, colors):
    """Return list of graph instances for the action of a 2x2 matrix on
    half of the unit circle.  That circle is broken into chunks each
    colored a different color.
      a, b, c, d  reals  entries of the matrix ul, ur, ll, lr
      colors  list of rgb tuples; len of this list is how many chunks
    (Terribly inefficient; runs through scatter_points many times)
    """
    r = []
    num_colors = len(colors)
    for i in range(num_colors):
        color = colors[i]
        points = find_angles(a,b,c,d,num_pts,
                             lower_limit=i*pi/num_colors,
                             upper_limit=(i+1)*pi/num_colors)
        g = scatter_plot(points,facecolor=color,edgecolor=color,
                         markersize=MARKERSIZE,ticks=TICKS)
        r.append(g)
    return r
```

Source of find_angles

This routine uses a formula for the angle between two vectors that always gives a positive value, that is, it is the angle without orientiation. That's fine for purpose here, which is to use the graph

to roughly locate places where the action of the matrix does not turn the vector.

```
def find_angles(a,b,c,d,num_pts,lower_limit=None,upper_limit=None):
    """Apply the matrix to points around the upper half circle, and
    return the angle between the input and output vectors.
      a, b, c, d  reals  Upper left, ur, ll, lr of matrix.
      num_pts  positive integer  number of points
      lower_limit=0, upper_limit=pi  ignore angles outside these limits
    """
    if lower_limit is None:
        lower_limit=0
    if upper_limit is None:
        upper_limit=pi
    r = []
    M = Matrix(RDF, [[a, b], [c, d]])
    for i in range(num_pts):
        t = i*pi/num_pts
        if ((t<lower_limit) or (t>upper_limit)):
            continue
        pt = (cos(t), sin(t))
        v = vector(RDF, pt)
        w = v*M
        try:
            angle = arccos(w*v/(1.0*w.norm()*v.norm()))
        except:
            angle = None
        r.append((t,angle))
    return r
```

Source of plot_before_after_action

The only perhaps unexpected point in this routine and its helper routine is that if the vector is not mapped very far then the helper routine does not show an arrow but instead shows a circle.

```
BA_THICKNESS = 1.5
def before_after_list(a, b, c, d, pts, colors=None):
    """Show the action of the matrix with entries a, b, c, d on the
    points by showning the vector before and the vector after in the
    same color.
      a, b, c, d  reals  Upper left, ur, ll, lr or matrix.
      pts  list of pairs of reals
      colors = None  list of colors
    """
    r = []
    for dex, pt in enumerate(pts):
        x, y = pt
        v = vector(RDF, pt)
        M = matrix(RDF, [[a, b], [c, d]])
        f_x, f_y = v*M
        if colors:
```

```
            color = colors[dex]
        else:
            color = 'lightgray'
        if ((abs(x-f_x) < EPSILON) and (abs(y-f_y) < EPSILON)):
            r.append(circle(pt, DOT_SIZE, color=color,
                            thickness=BA_THICKNESS))
        else:
            r.append(arrow((0,0), (x,y), color=color,
                     width=BA_THICKNESS, arrowsize=2*BA_THICKNESS))
            r.append(arrow((0,0), (f_x,f_y), color=color,
                     width=BA_THICKNESS, arrowsize=2*BA_THICKNESS))
    return r

def plot_before_after_action(a, b, c, d, pts, colors=None):
    """Show the action of the matrix with entries a, b, c, d on the
    points.
      a, b, c, d  reals  Upper left, ur, ll, lr or matrix.
      pt_list  list of pairs of reals; the before pts to show
    """
    if colors is None:
        colors = ["gray",]*len(pts)
    G =  Graphics()
    for ba in before_after_list(a,b,c,d,pts,colors=colors):
        G += ba
    p = plot(G)
    return p
```

Bibliography

Robert A. Beezer. Sage for Linear Algebra. http://linear.ups.edu/download/fcla-2.22-sage-4.7.1-preview.pdf, 2011.

S. J. Blank, Nishan Krikorian, and David Spring. A geometrically inspired proof of the singular value decomposition. *The American Mathematics Monthly*, pages 238–239, March 1989.

David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, March 1991.

Jim Hefferon. Linear Algebra. http://joshua.smcvt.edu/linearalgebra, 2012.

David Joyner and William Stein. Open source mathematical software. *Notices of the AMS*, page 1279, November 2007.

Python Team. Floating point arithmetic: issues and limitations, 2012a. URL http://docs.python.org/2/tutorial/floatingpoint.html. [Online; accessed 17-Dec-2012].

Python Team. The python tutorial, 2012b. URL http://docs.python.org/2/tutorial/index.html. [Online; accessed 17-Dec-2012].

Ron Brandt. *Powerful Learning*. Association for Supervision and Curriculum Development, 1998.

Sage Development Team. Sage tutorial 5.3. http://www.sagemath.org/pdf/SageTutorial.pdf, 2012a.

Sage Development Team. Sage reference manual 5.3. http://www.sagemath.org/pdf/reference.pdf, 2012b.

Shunryu Suzuki. *Zen Mind, Beginners Mind*. Shambhala, 2006.

Lloyd N. Trefethen and David Bau, III. *Numerical Linear Algebra*. Society for Industrial and Applied Mathematics, 1997.

Wikipedia. Extreme value theorem, 2012a. URL http://en.wikipedia.org/wiki/Extreme_value_theorem. [Online; accessed 28-Nov-2012].

Wikipedia. Lenna, 2012b. URL http://en.wikipedia.org/wiki/Lenna. [Online; accessed 29-Nov-2012].