# Lab Manual

for

Linear Algebra

by

Jim Hefferon

*Cover:* our Chocolate Lab, Suzy.

# Preface

This collection supplements the text *Linear Algebra*.[1] It helps students solidify and extend their understanding of the subject, using the mathematical software *Sage*.

A major goal of any undergraduate Mathematics program is to move students toward a higher-level grasp of the subject. At the start they take Calculus classes that relax completeness and rigor in favor of practicing algorithms, while later courses spend more effort on concepts and proofs. *Linear Algebra* fits into this developmental plan. It 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, by reinforcing the concepts.

Naturally the text uses examples and homework problems that are small 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 the same with twenty by twenty matrices of ten decimal place numbers would be badgering. However, an instructor can worry that this misses a chance to enhance students's understanding through explorations that are not hindered by the limitations of paper and pencil, or to make the point that the subject is widely applied, because in realistic examples of applications the computations are usually hard.

Mathematical software can spread the reach of what is reasonable to bigger systems, harder numbers, and longer computations. This manual extends what students can do in this direction. Besides helping solidify the understanding of the concepts that students have, an advantage of learning how to automate work and handle larger jobs is that this is more like what a professional must do in practice. Another advantage is that students see new ideas, such as runtime growth measures.

OK then, why not teach straight from the computer?

Our goal is to develop a higher-level understanding of the subject. For that, 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 find that their students are best served by sticking to the core material. Other instructors have students who will benefit from the kind of work that we do here. Because this manual is a supplement, it allows different teachers to make different choices.

[1] See hefferon.net/linearalgebra for a PDF that you can freely download, the ancillary materials, and the LaTeX source.

## Why *Sage*?

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

> 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 Janes 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 Janes result builds.

Professionals choose their tools by balancing many factors, but this argument is persuasive. Here we use *Sage* both because it is very capable and because it is Free, meaning not just that it is available at no cost but also that the source code is open for inspection by anyone.[2]

## This manual

This manual is Free. See hefferon.net/linearalgebra for the license. In particular, students are free to download it and instructors are free to redistribute it, such as on an intranet course web site.

Also Free are the software systems that we use to explore the subject, Python and *Sage*. Probably users already have them on their operating system but if not then they are available from www.python.org and www.sagemath.org. Both are powerful and capable software systems that have been in production use by thousands of professionals for years.

**A note on accuracy of the examples:** we often show input code for Python and *Sage*, along with the computer's responses. These are not cut and pasted. Instead, as part of generating this document, we grab the interaction. So what you see should be exact, unless your software version is very different from mine. This is my *Sage*.

```
sage: version()
SageMath version 9.0, Release Date: 2020-01-01
```

This is my Python.

```
>>> import sys
>>> print(sys.version)
3.8.10 (default, Jun  2 2021, 10:49:15)
[GCC 9.4.0]
```

[1] The full text is at www.ams.org/notices/200710/tx071001279p.pdf. [2] See www.gnu.org/philosophy/free-sw.html for some background and a definition.

## Reading this manual

Because this supplements the text *Linear Algebra*, we don't define the terms or prove the results here. Instead, a student should work through this material after covering the associated chapter in the book, using that for reference.

The association between what's here and the text is: *Python and Sage* does not depend on the book, *Gausss 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.)


## Acknowledgments

I am glad for this chance to thank the Python and *Sage* development teams. In particular, without [Sage Development Team, 2021b] 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 write this document.


*We emphasize practice.*
–Suzuki

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


Jim Hefferon
Mathematics, Saint Michael's College
Colchester, Vermont USA

2021-Oct-12

# Contents

# Python and *Sage*

*Sage* is based on Python so we'll start with that. Python is a popular language[1] with a simple style and powerful libraries. It is often used for scripting, as a glue to bring together separate parts.

This assumes that you already know some programming, so here you will just see enough Python to get started. For a more comprehensive introduction, work through Python's excellent tutorial, Python Team [2021b].

## Python basics

Start Python, for instance by giving the command `python3` in a terminal. You'll get a couple of lines of initial information followed by a prompt, three greater-than characters.

```
>>>
```

This is the Python interpreter. 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 in the interpreter. When you are done you can leave this prompt with ⟨*Ctrl*⟩-*D*.

Try entering these expressions. Python responds with the results shown (double star is exponentiation).

```
>>> 2 - (-1)
3
>>> 1 + 2*3
7
>>> 2**3
8
```

Part of Python's appeal is that simple things tend to be simple. Here is how you print something to the screen.[2]

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

As in other computer languages, variables give you a named place to keep values. The first line below puts 1 in the place called `i` and the second line involves fetching that 1.

[1] Written by a fan of Monty Python. [2] Python comes in two versions. Version 3 is what we use. Version 2 is quite old but you may still see it, for example if you look in a search engine for code. One difference between the two is that in version 2 the print statement did not have parentheses.

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

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

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

You can do multiple things on a single line.

```
>>> 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.

If you do something not allowed then Python complains, signaling its unhappiness by raising an exception and in the final line giving an error message.

```
>>> "a'' + 1
  File "<stdin>", line 1
    "a'' + 1
           ^
SyntaxError: EOL while scanning string literal
```

Besides integers, Python gives you real numbers and complex numbers.[1]

```
>>> 1/5
0.2
>>> 0.1 + 0.2
0.30000000000000004
>>> (3+2j) - (1-4j)
(2+6j)
```

As engineers do, Python uses the letter `j` for the square root of $-1$, not `i` as is traditional in Mathematics. (Although *Sage* lets you use `i`.)

Variables can also represent truth values.

[1] These operations are done with floating points, a system built into your computer's hardware that models the real numbers. In the example, the distinction leaks through since the decimal for 0.3 is not perfectly accurate. While issues brought out floating point representations are fascinating — see Python Team [2021a] and Goldberg [1991] — we shall ignore them, just to keep the focus on linear algebra.

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

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

Above we saw a couple of strings consisting of text between quotes. (They just had a single character, `'a'`, but Python doesn't worry about the difference between a character and a length one string.) To make a string, you can use either double or single quotes, as long as you use the same at both ends. Here `x` and `y` are double-quoted, which works because each contains an apostrophe, and the comma and period strings are single-quoted.

```
>>> 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."
```

The last line shows that `+` does string concatenation. It also shows that you can mix single-quoted strings with double-quoted ones.

To get a newline, you can use use slash-n, `\n`, inside a double-quoted string. If you need a lot of newlines then put explicit line breaks inside a string marked by three double quotes.

```
>>> 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 that start the lines after the first is Python's interpreter prompting you that what you have typed is not complete, that you've started a string but not yet finished it.

A Python *dictionary* is a finite function, a finite set of pairs $\langle key, value \rangle$, subject to the restriction that no key can appear twice. You can use a dictionary as a simple database.

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

Dictionaries are central to Python, in part because looking up values in a dictionary is very fast, due to the way that they are organized internally.

While dictionaries are made of pairs, a Python list is a sequence of single entries.

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

Lists can contain anything, including other lists.

```
>>> x = 4
>>> a = ['alpha', [True, x]]
>>> a
['alpha', [True, 4]]
>>> len(a)
2
```

Get an element from a list by specifying its index, its place in the list, inside square brackets. Lists indices are zero-based, that is, the initial element of the list is numbered 0.

```
>>> a = ['alpha', 'beta', 'gamma']
>>> a[0]
'alpha'
>>> a[1]
'beta'
>>> a[0] = 'Alpha'
>>> a[0]
'Alpha'
```

Count from the back by using negative indices.

```
>>> a = ['alpha', 'beta', 'gamma']
>>> a[-1]
'gamma'
```

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

```
>>> a = ['alpha', 'beta', 'gamma', 'delta']
>>> a[1:3]
['beta', 'gamma']
>>> a[1:-1]
['beta', 'gamma']
>>> a[1:1]
[]
```

You can lengthen a list.

```
>>> c = ['delta']
>>> c.append('epsilon')
>>> c
['delta', 'epsilon']
>>> a = ['beta', 'gamma']
>>> a.insert(0,'alpha')
>>> a+c
['alpha', 'beta', 'gamma', 'delta', 'epsilon']
```

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

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

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

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

One reason that non-mutability is useful is that tuples can be keys in dictionaries while lists cannot be keys.

```
>>> 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`, that means something has no sensible value. For instance, if your program keeps track of a person's address and includes a variable `apartment_num` then for a person who does not live in an apartment you should say `apartment_num = None`.

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

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

```
>>> delta
0
```

Here, Python sets `delta` to 1 if $x$ equals 0, otherwise Python sets `delta` to 0.

After `delta = 0` you must enter a blank line so that Python knows there is nothing more in that block. Notice also that double equals, `==`, means 'is equal to'. For 'not equal', use `!=`.

Python has two more variants on the `if` statement. It could have a single branch

```
>>> x = 0
>>> if (x == 0):
...     print("Division by x is not allowed.")
...
Division by x is not allowed.
```

or it could have more than two branches.

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

Computers excel at iteration, at looping through a sequence of statements. A `for` loop often involves a `range`.

```
>>> for i in range(4):
...     print(i, "squared is", i**2)
...
0 squared is 0
1 squared is 1
2 squared is 4
3 squared is 9
```

By default the lowest number produced by `range` is 0, which fits with the fact that list numbering is zero-based. Change the initial number by calling `range` with two arguments.

```
>>> for i in range(1, 4):
...     print(i, "cubed is", i**3)
...
1 cubed is 1
2 cubed is 8
3 cubed is 27
```

The highest number produced by `range` is one less than its input, so that the highest number given by `range(4)` or `range(1,4)` is 3. This is convenient because it makes the combination of two lists `range(4)+range(4,10)` give the same as the single list `range(10)`.

You can iterate over a list by working with element indices.

```
>>> 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
```

However, an experienced Python person who was not trying to illustrate the `range` command would replace `for i in range(len(x))` with `for c in x` because `for` can iterate over any sequence, not just a list of numbers. Here is a simple text formatter that fills a line with words until the line would exceed 40 characters, and then prints that line and starts a new one.

```
>>> quote = """Victorious warriors win first and then go to war,
... while defeated warriors go to war first and then seek to win.
... --Sun Tzu"""
>>> line, line_length = [], 0
>>> for wd in quote.split():
...     if line_length+len(wd) > 40:
...         print(" ".join(line))
...         line, line_length = [], 0
...     line_length = line_length + len(wd) + 1
...     line.append(wd)
...
Victorious warriors win first and then
go to war, while defeated warriors go to
war first and then seek to win. --Sun
>>> if line:
...     print(" ".join(line))
...
Tzu
```

Python has lot of string operations. Above, the `quote.split()` divides the string into separate words while `" ".join(line)` makes a string by putting space between the words in the list `line`. Thus, the final `if line:` above ejects any material left over when the `for` loop is finished.

A `for` loop is designed to execute a certain number of times. For a loop that runs a number of times that you don't know in advance, use `while`.

```
>>> n, i = 12, 0
>>> while (n != 1):
...     if (n % 2 == 0):
...         n = n // 2
...     else:
```

```
...         n = 3*n + 1
...     i = i + 1
...     print("after", i, "steps, n=", n)
...
after 1 steps, n= 6
after 2 steps, n= 3
after 3 steps, n= 10
after 4 steps, n= 5
after 5 steps, n= 16
after 6 steps, n= 8
after 7 steps, n= 4
after 8 steps, n= 2
after 9 steps, n= 1
```

The `//` operation divides the left integer by the right and then returns the floor.

Exit from a loop immediately with the `break` command.

```
>>> for i in range(10):
...     if (i == 3):
...         break
...         print("inside the loop: i is", i)
...
```

A common loop construct is to perform some action on each list element. Python has a shortcut for this, called list comprehension.

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

**Functions** You can make your own functions. This implements the quadratic formula.[1]

```
>>> def quad_formula(a, b, c):
...     """Find the roots of a quadratic
...       a, b, c  real numbers  Coefficients, as in ax^2 + bx + c
...     """
...     discriminant = (b**2 - 4*a*c)**(0.5)  # power 0.5 is square root
...     r1 = (-1*b-discriminant) / (2.0*a)
...     r2 = (-1*b+discriminant) / (2.0*a)
...     return (r1, r2)
...
>>> quad_formula(1, -6, 5)
(1.0, 5.0)
>>> quad_formula(1, 2, 1)
(-1.0, -1.0)
```

[1] This code is naive, for instance in that it does not address the potential floating point issues.

```
>>> quad_formula(1, -1, -1)
(-0.6180339887498949, 1.618033988749895)
```

When you write programs in Python, most of what you write is inside of functions.

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`.

About comments: the `quad_formula` function has two kinds of comments. Inside the code, inline comments start with a hash mark, `#` (programmers often write comments that take up an entire line, by starting that line with a hash). The second kind is that after the `def` line is a triple-quoted string that briefly describes what the function does and lists its parameters.

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

```
>>> def greet(name="Sir or Madam"):
...     """Say hello.
...       name  string  Person's name
...     """
...     print("Hello", name+".")
...
>>> greet("Fred")
Hello Fred.
>>> greet()
Hello Sir or Madam.
```

*Sage* heavily uses this feature of Python.

**Objects and modules** In Mathematics, a structure is a set of things that have associated operations. For example, a vector space is a set with the operations of addition and scalar multiplication. Python is object-oriented, which means that we can similarly bundle together data and actions (in this context the actions, the functions, are called methods).

```
>>> class DatabaseRecord(object):
...     """Hold information about a person.
...     """
...     def __init__(self, name, age):
...         self.name = name
...         self.age = age
...     def salutation(self):
...         print("Dear", self.name)
...
>>> a = DatabaseRecord("Jim", 62)
>>> a.name
'Jim'
>>> a.age
62
```

```
>>> a.salutation()
Dear Jim
>>> b = DatabaseRecord("Marie", 20)
>>> b.salutation()
Dear Marie
```

Above, we create two instances of `DatabaseRecord`, instance `a` and instance `b`. To get the age data of instance `a`, write `a.age`. (The `self` variable can be puzzling. Suppose that at the prompt you type `a.name="James"`. Then you've used the name `a` for the instance, so that the computer knows where to make the change. But inside the `class` description code there isn't any fixed instance name. That is, `self` refers to the current instance. For example, if you are working with the instance `a` then in the `salutation` the `self.name` refers to `a.name`.)

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 classes provided by other people. A library, a collection of related data, functions, and classes, in this context is called a module. For instance, Python comes with a `math` module that you can use.

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

The `import` statement makes the module's contents available.

Another module does random numbers.

```
>>> import random
>>> print(random.randint(1,10))
3
>>> print(random.randint(0,100)/100)
0.19
```

**Programs** The read-eval-print loop is great for small experiments but for code with more than a few lines, or for code that you want to keep, you want to put your work in a separate file. Here we will sketch how to write a stand-alone program.

To write the code, don't use a word processor. Instead, a text editor has the correct features for this. Use one with support for Python, including automatic indentation and syntax highlighting, where the editor colors your code to make it easier to read. There are many excellent editors; one is Emacs.[1]

Here is a first example of a Python program. Start your editor, open a new file called `month.py`, and enter these lines. (Notice that the first line describes the program in a comment. You may want to also name the author, the date, and the license for the code. In addition, the triple-quoted documentation string at the top of the file is a good practice. You should include such a string in each program.)

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

```
# month.py
"""Print the number of the current month. """
import datetime
current = datetime.datetime.now()  # get a datetime object, call its method
print("The month number is", current.month)
```

Run it through Python, One way is to open a command line terminal, change to the directory containing `month.py`, type `python3 month.py`, and hit ⟨*Enter*⟩. You should see output like `The month number is 10.`

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

```
# guessing_game.py
"""guessing_game.py

Guess the number between 1 and 10.
"""
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(input("Guess an integer between 1 and 10: "))
    flag = test_guess(guess)
```

(The `int` converts the string from `input` to an integer.) Here is output from running this game.

```
$ python3 guessing_game.py
Guess an integer between 1 and 10: 3
  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: 5
  You are right!
```

Note the triple-quoted documentation strings both for the file as a whole and for the function. Besides being useful to a programmer editing the file, they integrate with Python's help system.

Go to the directory containing `guessing_game.py` and start the Python interpreter. At the `>>>` prompt, enter `import guessing_game`. You will play through a round of the game (there is a way to avoid this but we will ignore the point). You are now using `guessing_game.py` as a module. Type `help("guessing_game")`. You will see documentation on `guessing_game`, including these lines (to see more than one page use the space bar and to quit the viewer use *q*).

```
Help on module guessing_game:

NAME
    guessing_game - guessing_game.py

DESCRIPTION
    Guess the number between 1 and 10.

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

Python got this information from the documentation strings. *Sage*'s programmers follow this practice and so you can use `help(...)` whenever you need more information on *Sage*'s routines.

## *Sage* basics

Learning about *Sage*'s capabilities is the goal of much of this manual. But *Sage* is built on Python and so a brief comparison here makes sense. See [Sage Development Team, 2021a] for a more broad-based introduction, and be aware of the reference [Sage Development Team, 2021b].

**Command line** *Sage*'s command line is like Python's but adapted to mathematical work. First start *Sage*, for instance by entering `sage` into a command line window. You get some initial text and a prompt.

```
sage:
```

Leave the prompt with the command `exit` followed by ⟨*Enter*⟩.

As with Python, you can use the interpreter to experiment.

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

```
sage: sin(pi/3).numerical_approx()
0.866025403784439
sage: sin(pi/3).n()
0.866025403784439
sage: n(sin(pi/3), digits=4)
0.8660
```

The final way shows that there is an option to limit the number of digits that you see.[1]

**Script** You can group *Sage* commands together in a file, to reuse them without having to retype. Create the file `sage_normal.sage`, enter this function, and save.

```
def normal_curve(upper_limit, digits=3):
    """Approximate area under the standard Normal curve from 0 to upper_limit.
         upper_limit  real  Find area from 0 to upperlimit.
    """
    mean, stddev = 0.0, 1.0
    area = numerical_integral((1/sqrt(2*pi) * e^(-0.5*((x-mean)/stddev)^2)),
                              0, upper_limit)
    return(numerical_approx(area[0], digits=digits))
```

Start *Sage* and bring the contents in with `load(...)`.

```
sage: load("normal_curve.sage")
None
sage: normal_curve(1.0)
0.341
```

**Notebook** In this manual we will stick to the interpreter. But we'll mention that another way to use *Sage* is in a browser-based graphical interface, inside a Jupyter Notebook, Jupyter Team [2021]. You can set up worksheets to use alone or with other people, you can easily go back and edit prior commands, view plots integrated with the text, and many other nice features. To fire it up, instead of just issuing the `sage` command, say `sage -n jupyter` (of course, you must have the Jupyter software installed).[2]

[1] If you don't use the option then you see the number of digits matching 53 bits of precision, which is the standard for floating point numbers. [2] If your operating system doesn't offer it then you can visit https://jupyter.org.

# Gauss's Method

*Sage* can solve linear systems in a number of ways. It can use a general system solver, and it can also use solvers specialized for linear systems. We'll see both.

## Systems of equations

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
```

As discussed in the prior chapter, here x is the name of a location in the computer's memory that we use to store and retrieve values.

Variables in equations are different; in the equation $C = 2\pi \cdot r$ the two variables do not have fixed values, nor are they tied to memory locations. To illustrate the difference, type in an unassigned variable such as `y`, and hit ⟨*Enter*⟩. You'll get an exception whose final line says `NameError: name 'y' is not defined.`

To instead use y as a symbolic variable, you must first declare it to the system.

```
sage: var('y')
y
sage: y
y
sage: 2*y
2*y
```

Earlier, *Sage* took x as a location to hold values and so it evaluated `7*x` (it got the value 21). But *Sage* does not evaluate `2*y` because you told it that 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]
```

Two things to note here: you must write double equals `==` in equations instead of the assignment

operator `=`, and you must write `2*x` instead of `2x`. Either mistake will trigger an error message saying `SyntaxError: invalid syntax`.

One way to solve this system is with *Sage*'s general-purpose solver.

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

That's pretty good but this solver is quite smart and can go beyond just numbers as answers. Next we put the parameter `a` in the right side. *Sage* solves for $x$, $y$, and $z$ in terms of that 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 there are better tools. We'll say more about matrices and vectors in later chapters but briefly: a *Sage* matrix is a list of rows, where each row is a list of numbers. We declare the kind of number at the start. In the book mostly we think of the entries as being real numbers but the examples and exercises use rational numbers, which are easier to read. *Sage* uses `QQ` for the set of rational numbers.

```
sage: M = matrix(QQ, [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
sage: M
[ 1  2  3  4]
[ 5  6  7  8]
[ 9 10 11 12]
sage: M.nrows()
3
sage: M.ncols()
4
sage: M[1,2]
7
```

As with Python lists, *Sage* lists are zero-indexed. 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* doesn't worry about the distinction between row and column vectors. There, `v` prints with rounded brackets, unlike matrices that print with square brackets. If you need to be careful about whether you have a row or column, turn the vector into a matrix.

```
sage: v = vector(QQ, [1,3,5])
sage: v.row()
[1 3 5]
sage: v.column()
[1]
[3]
[5]
```

The book often solves linear systems by augmenting 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]
```

An optional argument makes *Sage* show a distinction between the two parts of $M'$.

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

**Row operations** We can get *Sage* to do the arithmetic of Gauss's Method.

```
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 list indices start at zero, so the top row is row 0.

```
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.swap_rows(0,1)
None
sage: M_prime
[   2    0    4|   1]
[   0    2    1|   2]
[   2 -1/2    3|-1/2]
```

Next, rescale the top row by multiplying the entries by $1/2$.

```
sage: M_prime.rescale_row(0, 1/2)
None
sage: M_prime
[   1    0    2| 1/2]
[   0    2    1|   2]
[   2 -1/2    3|-1/2]
```

Replace the bottom row with the result of adding it to $-2$ times the top row.

```
sage: M_prime.add_multiple_of_row(2,0,-2)
None
sage: M_prime
[   1    0    2| 1/2]
[   0    2    1|   2]
[   0 -1/2   -1|-3/2]
```

Finally, replace the bottom row with the result of adding it and $1/4$ times the middle row.

```
sage: M_prime.add_multiple_of_row(2,1,1/4)
None
sage: M_prime
[   1    0    2| 1/2]
[   0    2    1|   2]
[   0    0 -3/4|  -1]
```

Now, by-hand back substitution would give the solution, or we can use *Sage*'s `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 above work in-place. That is, they change the matrix $M'$. *Sage* has related commands that leave the starting matrix unchanged, but return the changed matrix.

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

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

**Nonsingular and singular systems** The steps that we went through to reduce the matrix are mechanical. That is, we should be able to automate going from a matrix to its reduced row echelon form.

```
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]
```

So we have gotten the answer without *Sage*'s `solve`. Advantages of using methods specialized to linear systems are that they are faster, and can also be less prone to floating point issues.

In that example, M is nonsingular, because it is square and because when it has been put into echelon form, every column has a leading variable. The next example has a square matrix that is singular.

```
sage: M = matrix(QQ, [[1, 1, 1, 1], [1, 2, 3, 4], [2, 3, 4, 5], [0, 1, 2 ,3]])
sage: v = vector(QQ, [0, 1, 1, 1])
sage: M_prime = M.augment(v, subdivide=True)
sage: M_prime
[1 1 1 1|0]
```

```
[1 2 3 4|1]
[2 3 4 5|1]
[0 1 2 3|1]
sage: M_prime.rref()
[ 1  0 -1 -2|-1]
[ 0  1  2  3| 1]
[ 0  0  0  0| 0]
[ 0  0  0  0| 0]
```

Recall that if a linear system has a square matrix of coefficients that is singular then there are two possibilities. The first possibility is above, that in echelon form any row that is all zeros on the left has an entry on the right that is also zero. This 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 has no solution.

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

The difference between the possibilities has to do with the relationships among the rows of $M$ and also with the relationships among the entries of the vector. In the matrix the third row is the sum of the first two, and the fourth row is the difference of the first two. As to the vectors, in the first case the vector's entries have the same relationship — the third entry of the vector is the sum of the first and the fourth entry is the difference of the first two — while in the second case the vector's entries do not have that relationship.

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 associated homogeneous system.

```
sage: v = zero_vector(QQ, 4)
sage: v
(0, 0, 0, 0)
sage: M = matrix(QQ, [[1, 1, 1, 1], [1, 2, 3, 4], [2, 3, 4, 5], [0, 1, 2 ,3]])
sage: M_prime = M.augment(v, subdivide=True)
sage: M_prime
[1 1 1 1|0]
[1 2 3 4|0]
[2 3 4 5|0]
[0 1 2 3|0]
sage: M_prime.rref()
[ 1  0 -1 -2| 0]
[ 0  1  2  3| 0]
```

```
[ 0  0  0  0| 0]
[ 0  0  0  0| 0]
```

You can get the numbers of the columns having leading entries with the `pivots` method.

```
sage: M = matrix(QQ, [[1, 1, 1, 1], [1, 2, 3, 4], [2, 3, 4, 5], [0, 1, 2 ,3]])
sage: v = vector(QQ, [0, 1, 1, 1])
sage: M_prime = M.augment(v, subdivide=True)
sage: M_prime.rref()
[ 1  0 -1 -2|-1]
[ 0  1  2  3| 1]
[ 0  0  0  0| 0]
[ 0  0  0  0| 0]
sage: M_prime.pivots()
(0, 1)
sage: M_prime.nonpivots()
(2, 3, 4)
```

**Parametrization** Above we used *Sage*'s general purpose `solve` routine. It may not be the best way to find the solution of a linear system because specialized tools are likely to be faster and may be more accurate. But `solve` does come in handy to give the solution set of a system with infinitely many solutions, by parametrizing.

To illustrate, below we arrange for a system with infinitely many solutions. We start with a matrix of coefficients where the top two rows add to the bottom row and adjoin a vector with the same row relationship. We next find the reduced row echelon form. Then we transcribe the result to a system of equations and apply `solve`, in two different ways.

```
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.rref()
[ 1  0 -1| 2]
[ 0  1  2|-1]
[ 0  0  0| 0]
sage: var('x,y,z')
(x, y, z)
sage: eqns = [x+z == 2, y+2*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 to solve only for $x$ and $y$ and so *Sage* gives the solution in terms of $z$. In the second call *Sage* produces a parameter of its own.

## Automation

We finish by giving some Python code for two functions that mimic the steps that a person goes through in doing Gauss's Method by hand.

The source file of the script is below. The significance of "by hand" is that the script assumes that the matrices are small and have rational number entries. That means we don't have to worry about floating point issues. In short, this is a toy example that is fine for homework but is not ready for applications.

**Loading and running** First here are a few sample calls. Start *Sage* in the directory containing the file `gauss_method.sage`.

```
sage: load("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]
```

Besides `gauss_method`, the file also contains a `gauss_jordan` function to go all the way to reduced echelon form.

```
sage: load("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_jordan(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]
 take 2 times row 1
 take -1/17 times row 2
 take 17/32 times row 3
[     1      2      8|    -4]
[     0      0      1|-13/17]
[     0      0      0|     1]
 take 4 times row 3 plus row 1
 take 13/17 times row 3 plus row 2
[1 2 8|0]
[0 0 1|0]
[0 0 0|1]
 take -8 times row 2 plus row 1
[1 2 0|0]
[0 0 1|0]
[0 0 0|1]
```

**Source of** `gauss_method.sage` These are naive implementations of Gauss's Method and Gauss-Jordan reduction.

```
# Show Gauss's method and Gauss-Jordan reduction steps.
# 2012-Apr-20  Jim Hefferon  Public Domain.
# 2019-Nov-09 JH  Minor reformatting
# 2021-Sep-22 JH  Adjust for Python3

# 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 effects: M is reduced, steps are printed.
    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):
        # Do we 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 we are 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 effects: M is reduced, steps are printed.
    """
    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):
            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)
```

# Vector Spaces

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

In the book's chapter on vector spaces, the scalars come from the set of real numbers, $\mathbb{R}$. *Sage* lets you choose from two models of the real numbers. One is RDF, the computer's built-in floating point model of real numbers. The other is RR, which gives arbitrary precision reals. The first model runs faster and is more widely used in practice so that is what we will use.

## Real spaces

*Sage* lets you create vector spaces. Here is $\mathbb{R}^4$.

```
sage: V = VectorSpace(RDF,4)
sage: V
Vector space of dimension 4 over Real Double Field
```

You can also create subspaces. Here is a one-dimensional subspace of $\mathbb{R}^4$, described as the span of a set with one vector.

```
sage: V = VectorSpace(RDF,4)
sage: V
Vector space of dimension 4 over Real Double Field
sage: v1 = vector(RR, [1, 1, -3, 0])
sage: W = V.span([v1])
sage: W
Vector space of degree 4 and dimension 1 over Real Double Field
Basis matrix:
[ 1.0  1.0 -3.0  0.0]
```

You can ask *Sage* questions about this subspace, such as testing for membership.

```
sage: v3 = vector(RDF, [2, 2, -6, 0])
sage: v3 in W
True
sage: v4 = vector(RDF, [1, 0, 0, 5])
sage: v4 in W
False
```

Another example is $\mathbb{R}^3$.

```
sage: V = VectorSpace(RDF,3)
sage: V
Vector space of dimension 3 over Real Double Field
sage: v1 = vector(RDF, [1, 2, 3])
sage: v1 in V
True
sage: v2 = vector(RDF, [1, 2, 3, 4])
sage: v2 in V
False
```

We can create a subspace as a span of multiple vectors.

```
sage: V = VectorSpace(RDF,3)
sage: v1 = vector(RDF, [1, 0, 3])
sage: v2 = vector(RDF, [0, 1, 1])
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 3.0]
[0.0 1.0 1.0]
sage: v3 = vector(RDF, [1, 1, 4])
sage: v3 in W
True
```

**Basis** *Sage* will retrieve a basis for a vector space or subspace.

```
sage: V = VectorSpace(RDF,3)
sage: V.basis()
[
(1.0, 0.0, 0.0),
(0.0, 1.0, 0.0),
(0.0, 0.0, 1.0)
]
sage: v1 = vector(RDF, [2, -1, -3])
sage: v2 = vector(RDF, [1, 1, 0])
sage: W = V.span([v1, v2])
sage: W.basis()
[
(1.0, 0.0, -1.0),
(0.0, 1.0, 1.0)
]
```

Notice that the basis that *Sage* gives back is not the same as the set of two vectors $\{\vec{v}_1, \vec{v}_2\}$ that

we gave it. To get the new basis, *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 basis. Each matrix has one and only one reduced echelon form, so each subspace of $V$ has one and only one such basis; this is the canonical basis for the subspace.

*Sage* shows this basis when you ask for a description of the subspace.

```
sage: W
Vector space of degree 3 and dimension 2 over Real Double Field
Basis matrix:
[ 1.0  0.0 -1.0]
[ 0.0  1.0  1.0]
```

Another subspace of $\mathbb{R}^3$ is the plane described by the equation $x - 2y + 2z = 0$.

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

Here is that subspace.

```
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: W.basis()
[
(1.0, 0.0, -0.5),
(0.0, 1.0, 1.0)
]
```

If you take that set $\{\vec{v}_1, \vec{v}_2\}$ and add a vector $\vec{v}_3$ that is linearly independent of those two, then it generates a three dimensional subspace of $\mathbb{R}^3$, which is all of $\mathbb{R}^3$.

```
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),
(0.0, 0.0, 1.0)
]
```

If instead you add to the spanning set $\{\vec{v}_1, \vec{v}_2\}$ a third vector $\vec{v}_3$ that is linearly dependent on the other two then it doesn't change the span. That is, in this case the subspace spanned by the first two $[\vec{v}_1, \vec{v}_2]$ equals the span of all three $[\vec{v}_1, \vec{v}_2, \vec{v}_3]$.

```
sage: W = V.span([v1, v2])
sage: W.basis()
```

```
[
(1.0, 0.0, -0.5),
(0.0, 1.0, 1.0)
]
sage: v3 = vector(RDF, [0, 1, 1])
sage: W_prime = V.span([v1, v2, v3])
sage: W_prime.basis()
[
(1.0, 0.0, -0.5),
(0.0, 1.0, 1.0)
]
```

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

```
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)
]
```

**Equality** *Sage* can compare spaces. Here are two descriptions of the $xy$-plane inside of $\mathbb{R}^4$.

```
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])
```

If both vectors used to make the spanning set $W_{1,2}$ are members of $W_{1,3}$ then $W_{1,2} \subseteq W_{1,3}$. If in addition the vectors used to make the spanning set $W_{1,3}$ are members of $W_{1,2}$ then the spaces are equal.

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

Since both $\vec{v}_1, \vec{v}_3 \in W_{1,2}$ and $\vec{v}_1, \vec{v}_2 \in W_{1,3}$, the two subspaces are equal.

However, the more straightforward way to test equality is to just ask.

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

This exercise of the equality operator, ==, would be half-hearted without trying it on two spaces that are unequal.

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

There is a point here about algorithms. *Sage* could check for equality of two spans in a number of ways. One way is to check whether every member of the first spanning set is in the second space, and whether every member of the second spanning set is in the first space. But *Sage* does something different. It checks for equality of spaces just by checking whether they have equal canonical bases.

```
sage: W12.basis()
[
(1.0, 0.0, 0.0, 0.0),
(0.0, 1.0, 0.0, 0.0)
]
sage: W14.basis()
[
(1.0, 0.0, 0.0, 0.0),
(0.0, 1.0, 1.0, 1.0)
]
```

These two algorithms have the same external behavior, in that both decide whether two spaces are equal. But the algorithms differ internally and the second is faster, in part because *Sage* can pre-compute the canonical basis for a space and then use it as many times as desired. Finding the fastest way to do jobs is an important research area of computing.

**Operations** *Sage* can find 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)
]
```

If you try to intersect two spaces where the operation makes no sense, such as if you try to intersect $\mathbb{R}^3$ and $\mathbb{R}^4$, then *Sage* gives an error message whose final line contains the string `self and other must have the same ambient space.`

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

```
sage: V = VectorSpace(RDF,3)
sage: v1, v2 = vector(RDF, [1, 0, 0]), vector(RDF, [0, 1, 0])
sage: W12 = V.span([v1, v2])
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: V = VectorSpace(RDF,3)
sage: v1, v2 = vector(RDF, [1, 0, 0]), vector(RDF, [0, 1, 0])
sage: W12 = V.span([v1, v2])
sage: v3 = vector(RDF, [1, 1, 1])
sage: W3 = V.span([v3])
```

```
sage: W5 = W12 + W3
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]
sage: W5 == V
True
```

(The text covers the sum of subspaces in an optional section.)

## Other spaces

The book has vector spaces that aren't a subspace of some $\mathbb{R}^n$. To work with them, there are more sophisticated things that you can do but one straightforward thing is using a real space that is just like the desired one.[1] Consider this subspace of $\mathcal{P}_2$.

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

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

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

```
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 use *Sage* to perform computation for this space of $2\times 2$ matrices

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

by parametrizing

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

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

and then giving *Sage* the corresponding 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)
]
```

That example gets the vectors by reading across the matrix rows. You could instead read down the columns.

```
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. Much more on this is in the book's third chapter.

# Matrices

Matrix operations such as addition and multiplication are mechanical and therefore are perfectly suited for a computer.

## Constructing matrices

Define a matrix using *Sage*'s `matrix` function. Recall from the section on Gauiss's Method that it takes two inputs. First is the set of numbers describing which scalars can form the entries, and second is those entries, written as lists of rows. Here, for the first input we will use either `RDF` for floating point number entries, or `CDF` for complex number entries $a+bi$ where $a$ and $b$ are floating points, or we can fall back to the rational numbers, `QQ`.

```
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]
```

*Sage*'s default symbol for the square root of $-1$ is `i`. Because `i` is used for many things in programming, before using it for the complex numbers you should reset it with `CDF(i)`.

In this chapter, unless we have reason to do otherwise for scalars we will use the rational numbers, `QQ`, because rationals are easier to read — 1 is easier than 1.0 — and because the text's matrices usually have rational entries.

The `matrix` constructor lets 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]
```

One reason to do this is as a check on what you enter. Here the specified size doesn't match the entries because the given matrix has only two rows.

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

*Sage*'s error says `Number of rows does not match up with specified number.`

Another time that you want to list the number of rows and columns is when you are doing the following shortcut to get an identity matrix, putting a 1 in the place of the list of rows.

```
sage: I = matrix(QQ, 3, 3, 1)
sage: I
[1 0 0]
[0 1 0]
[0 0 1]
```

Much the same shortcut gets a zero matrix.

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

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 a command to create an identity matrix that can't lead to this error.

```
sage: I = identity_matrix(3)
sage: I
[1 0 0]
[0 1 0]
[0 0 1]
```

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

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

Still another example is that you can create a matrix by giving a pattern for the entries. Here in the created matrix the entry $a_{i,j}$ equals the sum $i+j$.

```
sage: A = matrix(QQ, 2, 3, lambda x, y: x+y)
sage: A
```

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

One last example: *Sage* will make a matrix with random entries, here with floating points from 0 to 1.

```
sage: random_matrix(RDF, 3, min=0, max=1)
[ 0.6037217481210441  0.09904843700474109   0.1281848272259799]
[ 0.1873893932980929   0.6823212453565581   0.7174872173884436]
[ 0.9842827932304326   0.9070107531197819   0.7690508010650767]
```

(Note the `RDF`. This is better for us because *Sage*'s `random_matrix` is more straightforward with floating points entries than with rational entries. The *Sage* reference has the details.)

## Linear combinations

Addition and subtraction of matrices are natural operations.

```
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* knows that adding matrices with different sizes is undefined.

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

You get an error whose final line contains `unsupported operand parent(s) for +`. In short, `+` is not defined between a $2\times 2$ matrix and a $2\times 3$ matrix.

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

```
sage: A = matrix(QQ, [[1, 2], [3, 4]])
sage: B = matrix(QQ, [[1, 1], [2, -2]])
sage: 3*A
[ 3  6]
[ 9 12]
sage: 3*A-4*B
[-1  2]
[ 1 20]
```

## Multiplication

**Matrix-vector product** Matrix-vector multiplication works the way that 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
([1 3 5 9]
[0 2 4 6], (1, 2, 3, 4))
sage: A*v
(58, 40)
```

The $2\times 4$ matrix $A$ multiplies the $4\times 1$ column vector $\vec{v}$ with the vector on the right side, as $A\vec{v}$. Trying this vector on the left, as below, gives `unsupported operand parent(s) for *`.

```
sage: v*A
```

Of course, you can multiply with a vector on the left if that vector's size suits the matrix. Here, the two-row $A$ works with a two-entry vector.

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

**Matrix-matrix product** *Sage* is happy to multiply matrices.

```
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 `unsupported operand parent(s) for '*'` since $BA$ is undefined.

Square matrices of the same size 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]
```

Note that the two results are different; matrix multiplication is not commutative.

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

In fact, matrix multiplication is very non-commutative in the sense that if you produce two $n \times n$ matrices at random then they almost surely don't commute. Here we multiply together a thousand random matrices to see if any commute.

```
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
```

Here is the result.

```
sage: number_commuting
0
```

**Inverse** If $A$ is an $n \times n$ nonsingular matrix then its inverse $A^{-1}$ is the $n \times n$ matrix such that $A^{-1}A = AA^{-1}$ is the $n \times n$ identity matrix. In the book, while we use a formula for do $2 \times 2$ inverses, to find larger inverses we write the original matrix next to the identity and then perform Gauss-Jordan reduction.

```
sage: A = matrix(QQ, [[1, 3, 1], [2, 1, 0], [4, -1, 0]])
sage: A.is_singular()
False
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 on the right. To pull out the inverse, *Sage* has a `matrix_from_columns` method on a matrix instance that takes a list of columns, extracts them, and returns the matrix composed of those columns.

```
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_inv*A
```

```
[1 0 0]
[0 1 0]
[0 0 1]
sage: A*A_inv == identity_matrix(3)
True
```

All this is very awkward, so naturally there is a standalone command.

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

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

have the same matrix of coefficients on the left sides but different right sides. If you calculate the inverse of that matrix then solving each system requires only a single 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)
```

## Condition number

For the by-hand linear systems in the text, we find exact solutions. But in applications we use floating points and so the computation may lose precision because of numerical issues.

Some systems are more subject to this loss than others. The suceptibility of a system's matrix of coefficients to this problem is measured by its *condition number*. This is a positive real number that estimates worst-case loss of precision.[1]

[1] The condition number's base 10 logarithm is a worst-case estimate of how many digits are lost in solving a linear system with that matrix of coefficients. So a condition number is large if its base 10 logarithm is greater than or equal to the number of significant digits of the entries in the matrix.

More precisely, we start with a linear system $A\vec{v} = \vec{d}$ and then ask how much an inaccuracy in $\vec{d}$ (perhaps due to some floating point issue or perhaps some measurement limit on the vector's entries) can affect the solution $\vec{v}$. That is, we are considering $A(\vec{v} + \Delta\vec{v}) = \vec{d} + \Delta\vec{d}$ and asking for the relationship between $\Delta\vec{d}$ and $\Delta\vec{v}$. Linearity gives $A\Delta\vec{v} = \Delta\vec{d}$ so we are asking: how much is $\Delta\vec{v}$ affected by $\Delta\vec{d}$, via the mediation of the matrix $A$?

Define the *2-norm* of a vector to be its length, $\|\vec{v}\| = \sqrt{v_1^2 + \cdots + v_n^2}$ (there are many norms but this is the most common).

```
sage: v = vector(RDF, [1, 2, 3])
sage: v.norm(2)
3.7416573867739413
```

There is some input vector that $A$ stretches the most, a vector $\vec{v}_{max}$ where the number $M = \|A\vec{v}_{max}\|/\|\vec{v}_{max}\|$ is maximal. There is also a vector that $A$ shrinks the most, a vector $\vec{v}_{min}$ where the number $m = \|A\vec{v}_{min}\|/\|\vec{v}_{min}\|$ is a minimum (supposing that $\vec{v}_{min} \neq \vec{0}$).[1] The condition number is the ratio of the two, $\kappa(A) = M/m$, except that because a singular matrix maps some nonzero vectors to zero and so has $m = 0$, in this case we set $\kappa(A) = \infty$.

The definition of $M$ gives $\|\vec{d}\| \leq M\|\vec{v}\|$ and the definition of $m$ gives $\|\Delta\vec{d}\| \geq m\|\Delta\vec{v}\|$. Thus the relative change in the system's computed solution and the relative change in the system's right side compare in this way.

$$\frac{\|\Delta\vec{v}\|}{\|\vec{v}\|} \leq \kappa(A) \cdot \frac{\|\Delta\vec{d}\|}{\|\vec{d}\|}$$

That is, the condition number is the worst-case factor by which the relative error is magnified. Changes in the right side can result in changes in the computed solution that are $\kappa(A)$ times as large.

```
sage: A = matrix(RDF, 3, [[1, 2, 3], [4, 5, 6], [7, 8, 9.001]])
sage: A.condition(2)
101072.26488077441
```

Matrices that are nearly singular have a higher condition number than those that are far from singular.

```
sage: v = []
sage: for i in range(300):
....:     A = random_matrix(RDF, 3, min=0, max=100)
....:     v.append(A.condition(2))
```

Now we can get an idea of what is typical.

```
sage: mean(v), std(v)
(40.908286661714826,  115.37021561052416)
```

If the condition number is large then we say that the system for the problem is 'ill-conditioned', otherwise we say it is 'well-conditioned'.

[1] We discuss these factors more in the next chapter.

## Running time

Large linear algebra problems occur frequently in science and engineering. Since computers are fast and accurate, they allow us to solve problems that we could not hope to do by hand.

But even with computers, there are limits. One limit on just how large a problem we can do is how quickly the machine can find the answer. Naturally, problems with larger matrices tend to take longer to solve. Comparing the size of a problem to the time that an algorithm takes to solve it is an important way to gauge the usefulness of that algorithm.

The matrix inverse operation is a good illustration. (The entries in these matrices are floating points because these are the most common in applications.)

```
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: 62.7 s per loop
```

*Sage*'s `timeit` tells you how long an operation takes. It does not report just the time that the CPU spent but rather how long the operation took if you timed it with a the clock on the wall. It runs the command a number of times, to mitigate against the computer being slowed down by a disk write or other interruption. The bottom line says it did three batches of running the command 625 times each and the average time in the fastest batch was 62.7 microseconds.[1] That's fast, but then $A$ is only $3\times 3$.

And what's more, $A$ is a particular $3\times 3$ matrix. For an estimate of how long it typically takes, 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: 86 s per loop
sage: timeit('random_matrix(RDF, 3, min=-1, max=1).inverse()')
625 loops, best of 3: 85.8 s per loop
sage: timeit('random_matrix(RDF, 3, min=-1, max=1).inverse()')
625 loops, best of 3: 86.6 s per loop
```

One issue with this performance data is that we can't tell how much of the time is spent generating the random matrix and how much is spent finding the inverse. The code below takes some sizes, $3\times 3$, $10\times 10$, etc., finds a single random matrix and then gets the time to compute the inverse of that matrix.

```
sage: for size in [3, 10, 25, 50, 75, 100, 150, 200]:
....:     print("size = "+str(size))
....:     M = random_matrix(RR, size, min=-1, max=1)
....:     timeit('M.inverse()')
....:
```

[1] This machine identifies as: Intel(R) Core(TM) i7-6820HQ CPU @ 2.70GHz.

```
size = 3
625 loops, best of 3: 55 s per loop
size = 10
625 loops, best of 3: 548 s per loop
size = 25
125 loops, best of 3: 6.79 ms per loop
size = 50
5 loops, best of 3: 53 ms per loop
size = 75
5 loops, best of 3: 177 ms per loop
size = 100
5 loops, best of 3: 419 ms per loop
size = 150
5 loops, best of 3: 1.41 s per loop
size = 200
5 loops, best of 3: 3.4 s per loop
```

Some of those times are in microseconds, some are in milliseconds, and some are in seconds. A microsecond is one-millionth of a second, 0.000 001 seconds. A millisecond is a thousandth of a second, 0.001 seconds. This table is consistently in seconds.

| *size* | *seconds* |
|---|---|
| 3 | 0.000 055 |
| 10 | 0.000 548 |
| 25 | 0.006 79 |
| 50 | 0.053 |
| 75 | 0.177 |
| 100 | 0.419 |
| 150 | 1.41 |
| 200 | 3.4 |

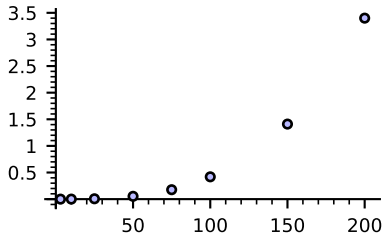
The time grows faster than the size. For instance, doubling the size from 25 to 50 increases the time by more than two: 0.053/0.00679 is about 7.8. Similarly, increasing the size four-fold from 50 to 200 causes the time to increase by much more than a factor of four: $3.4/0.053 \approx 64.15$. And going from a problem size of 10 to a size of 100 increases the time taken by a factor of $0.419/0.000\,548 \approx 764.60$.

Get a graph by giving *Sage* the data as a list of pairs.[1]

```
sage: d = [(3, 0.000055), (10, 0.000548), (25, 0.00679),
....:      (50, 0.053), (75, 0.177), (100, 0.419),
....:      (150, 1.41), (200, 3.4)]
sage: g = scatter_plot(d)
sage: g.save("graphics/mat001.pdf")
None
```

[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 decoration code as clutter.

(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.)




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. A caution if you are trying this yourself: to generate this data, `timeit` took so long that the computer had to be left to run overnight.

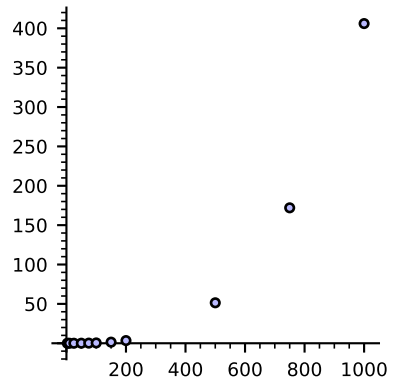
```
sage: for size in [500, 750, 1000]:
....:         print "size=",size
....:     M = random_matrix(RR, size, min=-1, max=1)
....:     timeit('M.inverse()')
....:
size= 500
5 loops, best of 3: 51.4 s per loop
size= 750
5 loops, best of 3: 172 s per loop
size= 1000
5 loops, best of 3: 406 s per loop
```

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

| *size* | *seconds* |
|---|---|
| 500 | 51.4 |
| 750 | 172. |
| 1000 | 406. |

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

```
sage: d = [(3, 0.000055), (10, 0.000548), (25, 0.00679),
....:      (50, 0.053), (75, 0.177), (100, 0.419),
....:      (150, 1.41), (200, 3.4)]
sage: d = d + [(500, 51.4), (750, 172), (1000, 406)]
sage: g = scatter_plot(d)
sage: g.save("graphics/mat002.pdf")
None
```




(Note that the two graphs have different scales; if the graph on this page had the same vertical scale as the one on the prior page then it would extend far off the top of the paper.) So a practical limit to the size of a problem that we can solve with the matrix inverse operation comes from the fact that the graph above is not a line — the time required grows much faster than the size.

A major effort in Computer Science is to find fast algorithms to do practical tasks. Many people are working 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 functions between vector spaces.

As we've done earlier, although we think of the vector spaces as having real number scalars, here we shall define matrices with rational number entries, just because they illustrate the points perfectly well and are easier to read.

## Linear functions

*Sage* can work with functions.

$$t(\begin{pmatrix} a \\ b \end{pmatrix}) = \begin{pmatrix} a+2b \\ a+2b \end{pmatrix}$$

```
sage: a, b = var('a, b')
sage: t_symbolic(a, b) = [a+2*b, a+2*b]
sage: t_symbolic
(a, b) |--> (a + 2*b, a + 2*b)
sage: t_symbolic(1,3)
(7, 7)
```

*Sage* can find a scalar multiple of a function, or add two functions.

```
sage: t_symbolic(a, b) = [a+2*b, a+2*b]
sage: f_symbolic = 3 * t_symbolic
sage: f_symbolic
(a, b) |--> (3*a + 6*b, 3*a + 6*b)
sage: f_symbolic(4, 2)
(24, 24)
sage: s_symbolic(a,b) = [a+b, a-b]
sage: g_symbolic = s_symbolic + t_symbolic
sage: g_symbolic
(a, b) |--> (2*a + 3*b, 2*a + b)
sage: g_symbolic(1, 5)
(17, 7)
```

*Sage* can also find the composition of two functions.

```
sage: s_symbolic * f_symbolic
(a, b) |--> 3*(a + 2*b)*(a + b) + 3*(a + 2*b)*(a - b)
sage: s_symbolic * f_symbolic (1,2)
(a, b) |--> 30*a
```

We used the oddball names, `t_symbolic`, `s_symbolic`, etc., because these are not functions in that we have not specified a domain and codomain; rather they are prototypes for functions. We can use one to produce a function, a linear function $t\colon \mathbb{R}^2 \to \mathbb{R}^2$.

```
sage: a, b = var('a, b')
sage: t_symbolic(a, b) = [a+2*b, a+2*b]
sage: t = linear_transformation(QQ^2, QQ^2, t_symbolic)
```

*Sage* uses the term 'linear transformation' for any linear map at all, which is a common usage, while the book restricts it to only maps where the domain and codomain are equal.

With the transformation t we expect this

$$t(\begin{pmatrix}1\\3\end{pmatrix}) = \begin{pmatrix}7\\7\end{pmatrix}$$

and *Sage* delivers.

```
sage: v = vector(QQ, [1, 3])
sage: t(v)
(7, 7)
```

## Left/right

By default, *Sage* represents linear maps differently than the book does. An example explains it best. Consider again the transformation $t\colon \mathbb{R}^2 \to \mathbb{R}^2$.

$$t(\begin{pmatrix}a\\b\end{pmatrix}) = \begin{pmatrix}a+2b\\a+2b\end{pmatrix}$$

To represent it with respect to $B, D \subset \mathbb{R}^2$, we can use the canonical bases $B = \mathcal{E}_2$, $D = \mathcal{E}_2$. We find the effect of t on the elements of B and represent the results with respect to D.

$$\begin{pmatrix}1\\0\end{pmatrix} \xmapsto{t} \begin{pmatrix}1\\1\end{pmatrix} \text{ gives } \mathrm{Rep}_D(\begin{pmatrix}1\\1\end{pmatrix}) = \begin{pmatrix}1\\1\end{pmatrix} \qquad \begin{pmatrix}0\\1\end{pmatrix} \xmapsto{t} \begin{pmatrix}2\\2\end{pmatrix} \text{ gives } \mathrm{Rep}_D(\begin{pmatrix}2\\2\end{pmatrix}) = \begin{pmatrix}2\\2\end{pmatrix}$$

Thus the book gets this matrix.

$$\mathrm{Rep}_{B,D}(t) = \begin{pmatrix}1 & 2\\1 & 2\end{pmatrix}$$

However, *Sage* shows a different one.

```
sage: t_symbolic(a, b) = [a+2*b, a+2*b]
sage: t = linear_transformation(QQ^2, QQ^2, t_symbolic)  # see discussion of 'side' below
```

```
sage: t
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
```

What's happening?

Remember that the point of representations is to use the matrix representation of t and the vector representation of $\vec{v}$ to compute the vector representation of $t(\vec{v})$. The book writes

$$\mathrm{Rep}_{B,D}(t)\,\mathrm{Rep}_B(\vec{v}) = \mathrm{Rep}_D(t(\vec{v}))$$

with the vector on the right of the matrix. But *Sage* by default takes the representation vector on the left. Obviously the difference is cosmetic — the translation is that, compared to the book's $T\vec{v}$, *Sage*'s default is $\vec{v}^{\,T}\,T^T$ — and that's why *Sage*'s matrix is the transpose of the book's matrix.

To have *Sage* conform to the book's convention we will do two things: define the map with the option `side='right'`, and show its matrix with the same option.

```
sage: t_symbolic(a, b) = [a+2*b, a+2*b]
sage: t = linear_transformation(QQ^2, QQ^2, t_symbolic, side='right')
sage: t.matrix(side='right')
[1 2]
[1 2]
```

## Defining a linear map

We will describe two ways to define a linear map.

**Symbolically** We've already introduced defining a map by formula.

```
sage: a, b = var('a, b')
sage: h_symbolic(a, b) = [a+b, a-b, b]
sage: h_symbolic
(a, b) |--> (a + b, a - b, b)
sage: h = linear_transformation(QQ^2, QQ^3, h_symbolic, side='right')
sage: h.matrix(side='right')
[ 1  1]
[ 1 -1]
[ 0  1]
```

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

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

If the vector isn't a member of the domain

```
sage: v1 = vector(QQ, [1,2,3,4])
sage: h(v1)
```

then *Sage* gives a long traceback error message, but as usual the final line is the most informative since it contains: `Type Error:(1, 2, 3, 4) fails to convert into the map's domain.`

*Sage* can find the null space and range space, using the terms 'kernel' and 'image'.

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

*Sage* has found that the null space is the trivial subspace of the domain, $\mathscr{N}(\mathrm{h}) = \{\vec{0}\}$. This is a 0-dimensional subspace so *Sage* reports that its basis is empty. *Sage* found that the range space $\mathscr{R}(\mathrm{h})$ is 2-dimensional, and gives its basis.

Because the null space is trivial, the map is one-to-one. Because the range space is 2-dimensional while the codomain is 3-dimensional, the map is not onto. Note that these findings fit with the theorem that the dimension of the null space plus the dimension of the range space, which are 0 and 2, equals the dimension of the domain.

As another example, consider this function.

```
sage: t_symbolic(a, b) = [a+2*b, a+2*b]
sage: t_symbolic
(a, b) |--> (a + 2*b, a + 2*b)
sage: t = linear_transformation(QQ^2, QQ^2, t_symbolic, side='right')
```

This is not one-to-one since there are two inputs that map to the same output.

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

Another way to tell that the map is not one-to-one is compute that $\mathscr{N}(\mathrm{t})$ is not 0-dimensional.

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

Before looking at this map's range space, we know that the dimension of that range must be 1 because the dimensions of the null space and range space add to the dimension of the domain.

*Sage* confirms.

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

**Via matrices** The other way that we can define a linear map is with a matrix.

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

Because the matrix has rational entries, *Sage* takes the domain and codomain to be $\mathbb{Q}^2$ and $\mathbb{Q}^3$. Again: by default *Sage* prefers the representation where the vector multiplies from the left so we specify `side='right'`.

```
sage: m
Vector space morphism represented by the matrix:
[1 3 5]
[2 4 6]
Domain: Vector space of dimension 2 over Rational Field
Codomain: Vector space of dimension 3 over Rational Field
```

Although we defined the linear map `m` with `side='right'`, the matrix that *Sage* would show by default is for `side='left'`. To get the representation fitting the text, ask for it explicitly.

```
sage: m.matrix(side='right')
[1 2]
[3 4]
[5 6]
```

When defined in this way, *Sage* takes the linear map to be the one represented by the matrix with respect to the standard basis.

```
sage: m = linear_transformation(M, side='right')
sage: v = vector(QQ, [7, 8])
sage: v
(7, 8)
sage: m(v)
(23, 53, 83)
```

(If you left `side='right'` out of the definition of the linear map and then multiplied by the two-element vector then you would get an error message complaining that the vector is not in the map's 3-dimensional domain.)

Although the maps below are defined in two ways, *Sage* can mix them together.

```
sage: M = matrix(QQ, [[1, 2], [3, 4], [5, 6]])
sage: m = linear_transformation(M, side='right')
sage: n_symbolic(a, b) = [a+2*b, 3*a+4*b, 5*a+6*b]
sage: n = linear_transformation(QQ^2, QQ^3, n_symbolic, side='right')
sage: m == n
True
```

And, you can ask the same questions of linear maps created from matrices that we asked of linear maps created from symbolic functions.

```
sage: M = matrix(QQ, [[1, 2], [3, 4], [5, 6]])
sage: m = linear_transformation(M, side='right')
sage: m.kernel()
Vector space of degree 2 and dimension 0 over Rational Field
Basis matrix:
[]
```

You can also define a map by starting with a matrix that represents the map with respect to nonstandard bases. Here we define nonstandard bases for both the domain and codomain, $B = \langle \vec{\beta}_1, \vec{\beta}_2 \rangle$ and $D = \langle \vec{\delta}_1, \vec{\delta}_2 \rangle$.

```
sage: M = matrix(QQ, [[1, 2], [3, 4]])
sage: beta_1 = vector(QQ, [1, -1])
sage: beta_2 = vector(QQ, [1, 1])
sage: domain_basis = [beta_1, beta_2]
sage: D = (QQ^2).subspace_with_basis(domain_basis)
sage: delta_1 = vector(QQ, [2, 0])
sage: delta_2 = vector(QQ, [0, 3])
sage: codomain_basis = [delta_1, delta_2]
sage: C = (QQ^2).subspace_with_basis(codomain_basis)
sage: m = linear_transformation(D, C, M, side='right')
sage: m.matrix(side='right')
[1 2]
[3 4]
sage: m(vector(QQ, [1, 0]))
(3, 21/2)
```

*Sage* has calculated that

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

and used it to compute the image vector.

$$\begin{pmatrix}1&2\\3&4\end{pmatrix}\begin{pmatrix}1/2\\1/2\end{pmatrix} = \begin{pmatrix}3/2\\7/2\end{pmatrix} = \text{Rep}_D(m(\vec{v})) \quad \text{so} \quad m(\vec{v}) = (3/2)\begin{pmatrix}2\\0\end{pmatrix} + (7/2)\begin{pmatrix}0\\3\end{pmatrix} = \begin{pmatrix}3\\21/2\end{pmatrix}$$

## Operations

Fix some vector space domain D and codomain C and consider the set of all linear transformations between them. This collection has some natural operations, including addition and scalar multiplication. Earlier we saw that *Sage* can work with these operations as well as composition, and we close by showing the effect that they have on the representations.

**Addition and scalar multiplication** 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, side='right')
sage: N = matrix(QQ, [[5, -1], [0, 7]])
sage: n = linear_transformation(QQ^2, QQ^2, N, side='right')
sage: m.matrix(side='right')
[1 2]
[3 4]
sage: n.matrix(side='right')
[ 5 -1]
[ 0  7]
sage: (m+n).matrix(side='right')
[ 6  1]
[ 3 11]
```

The parentheses in the final line are there because if we enter `m+n.matrix(side='right')` then *Sage* tries to combine the linear map `m` with the matrix `n.matrix(side='right')`.

Similarly, scalar multiplication of a linear map is reflected in the scalar multiplication of the matrix.

```
sage: M = matrix(QQ, [[1, 2], [3, 4]])
sage: m = linear_transformation(QQ^2, QQ^2, M, side='right')
sage: (3*m).matrix(side='right')
[ 3  6]
[ 9 12]
sage: (m*3).matrix(side='right')
[ 3  6]
[ 9 12]
```

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

```
sage: M = matrix(QQ, [[1, 2], [3, 4]])
sage: m = linear_transformation(QQ^2, QQ^2, M, side='right')
sage: N = matrix(QQ, [[5, -1], [0, 7]])
sage: n = linear_transformation(QQ^2, QQ^2, N, side='right')
```

```
sage: M*N
[ 5 13]
[15 25]
```

As the book emphasizes, the point of matrix multiplication is that it represents linear map composition.

```
sage: (m*n).matrix(side='right')
[ 5 13]
[15 25]
```

# Singular Value Decomposition

We will picture how some linear transformations $t\colon \mathbb{R}^2 \to \mathbb{R}^2$ act (we use $\mathbb{R}^2$ because the pictures are easy). We will then extend the ideas to higher dimensions, and give a practical consequence.

## Drawing the action

A defining property of linear maps is that $h(r \cdot \vec{v}) = r \cdot h(\vec{v})$. Recall that a line through the origin in $\mathbb{R}^n$ has the form $\{r \cdot \vec{v} \mid r \in \mathbb{R}\}$ for some $\vec{v} \neq \vec{0}$. Thus the scalar multiplication property imposes a uniformity condition on a linear map: its action on any line through the origin is determined by its action on any one of the nonzero vectors in that line.

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

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

and consider the transformation $t\colon \mathbb{R}^2 \to \mathbb{R}^2$ that is represented by this matrix.

$$\mathrm{Rep}_{\mathcal{E}_2,\mathcal{E}_2}(t) = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

If you pick a domain vector $\vec{v}$ then computing the effect of $t$ is easy.

$$\vec{v} = \begin{pmatrix} 1 \\ 2 \end{pmatrix} \overset{t}{\longmapsto} \begin{pmatrix} 5 \\ 11 \end{pmatrix}$$

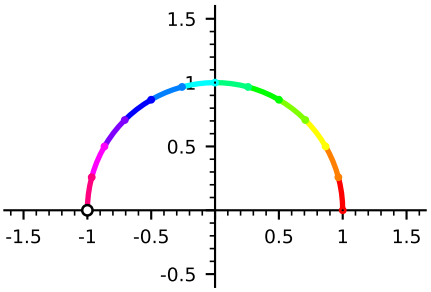
The prior paragraph notes that on $2\vec{v}$ the map has twice the effect, on $3\vec{v}$ it has three times the effect, etc.

$$\begin{pmatrix} 2 \\ 4 \end{pmatrix} \overset{t}{\longmapsto} \begin{pmatrix} 10 \\ 22 \end{pmatrix} \qquad \begin{pmatrix} 3 \\ 6 \end{pmatrix} \overset{t}{\longmapsto} \begin{pmatrix} 15 \\ 33 \end{pmatrix} \qquad \begin{pmatrix} r \\ 2r \end{pmatrix} \overset{t}{\longmapsto} \begin{pmatrix} 5r \\ 11r \end{pmatrix}$$

So, to picture a map's action on the whole plane, we will fix one nonzero vector $\vec{v}$ from each line through the origin and show where the transformation takes it. The natural set containing one point from each line through the origin is the upper half unit circle, shown here (the colors are

explained below).

$$U = \{\vec{v} = \begin{pmatrix} \cos(t) \\ \sin(t) \end{pmatrix} \mid 0 \leq t < \pi\}$$




To have exactly one representative of the $x$ axis, the point $(1, 0)$ is included in that set but $(-1, 0)$ is not.

The first transformation that we will illustrate is simple.

$$\begin{pmatrix} x \\ y \end{pmatrix} \overset{t}{\longmapsto} \begin{pmatrix} 2x \\ y \end{pmatrix}$$

The code below loads `plot_action.sage` to get access to `plot_circle_action(a,b,c,d)`. This divides the input upper half circle into a number of parts, the red one, the orange one, etc., and then it multiplies points on those curved segments by this matrix

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

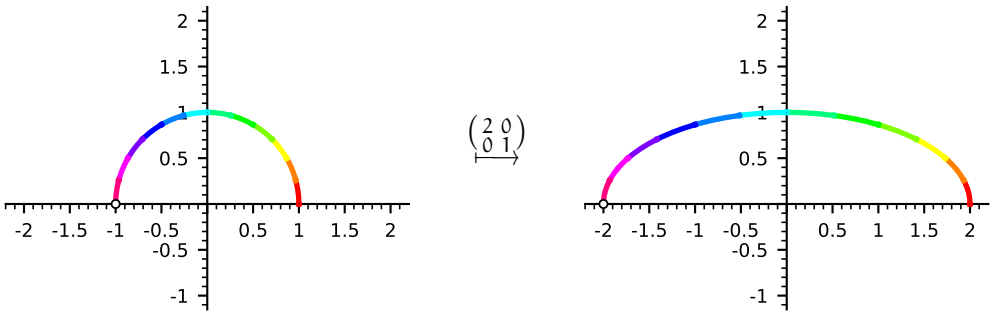
(for this first transformation we take $a = 1$, $b = 0$, $c = 0$, and $d = 2$). Finally it connects those output segments to get an output curve, where the output segments are colored red, orange, etc. The listing of the full source of `plot_action.sage` is at the end of this chapter.

To show before and after pictures, the code first plots the circle unchanged, that is, under the effect of the identity map, and then plots the result of applying the transformation.

```
sage: load("plot_action.sage")
None
sage: q = plot_circle_action(1,0,0,1)
sage: q.set_axes_range(-2, 2, -1, 2)
None
sage: q.save("graphics/svd001a.pdf")
None
sage: p = plot_circle_action(2,0,0,1)
sage: p.set_axes_range(-2, 2, -1, 2)
None
sage: p.save("graphics/svd001b.pdf")
None
```

Here are the before and after pictures. The colors are there to show the association — to show

which points on the left are mapped to which points on the right.



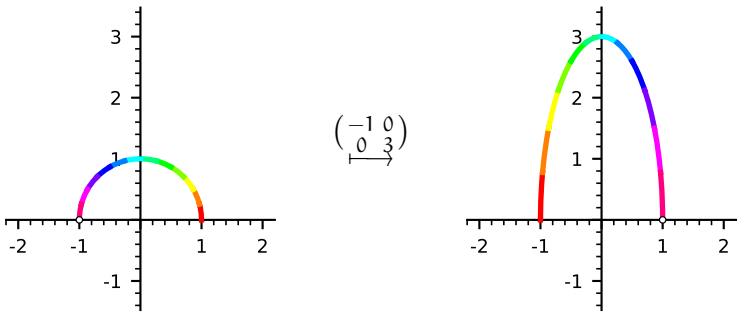

The effect of this map is to stretch things in the x direction by a factor of two.

We next show the action of the map

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

that triples the y component and also multiplies the x component by $-1$.[1]

```
sage: q = plot_circle_action(1,0,0,1)
sage: q.set_axes_range(-2, 2, -1.25, 3.25)
None
sage: q.save("graphics/svd002a.pdf")
None
sage: p = plot_circle_action(-1,0,0,3)
sage: p.set_axes_range(-2, 2, -1.25, 3.25)
None
sage: p.save("graphics/svd002b.pdf")
None
```



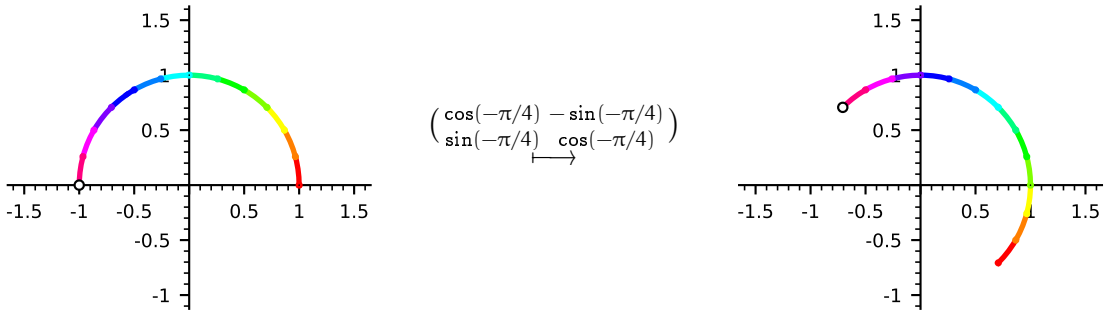

Tripling the y component is easy to understand. As for taking the negative of the x component, this is where the colors are especially useful. On the input circle, when you move counterclockwise, the colors go from red to orange, to green, blue, indigo, and then violet. But the output does the opposite: moving counterclockwise it passes from violet to red. This transformation changes the orientation, or 'sense', of the curve.

[1] As in other chapters, some of the graphics are drawn using some options that are not shown. In this case, the two `p.save(...)` commands have the `ticks_integer=True` option. Not showing these options just reduces clutter that isn't linear algebra.

The next transformation is rotation by $\pi/4$ radians, clockwise.

$$\begin{pmatrix} x \\ y \end{pmatrix} \mapsto \begin{pmatrix} \cos(-\pi/4)\cdot x - \sin(-\pi/4)\cdot y \\ \sin(-\pi/4)\cdot x + \cos(-\pi/4)\cdot y \end{pmatrix}$$

```
sage: q = plot_circle_action(1,0,0,1)
sage: q.set_axes_range(-1.5, 1.5, -1, 1.5)
None
sage: q.save("graphics/svd003a.pdf")
None
sage: p = plot_circle_action(cos(-pi/4),-sin(-pi/4),sin(-pi/4),cos(-pi/4))
sage: p.set_axes_range(-1.5, 1.5, -1, 1.5)
None
sage: p.save("graphics/svd003b.pdf")
None
```



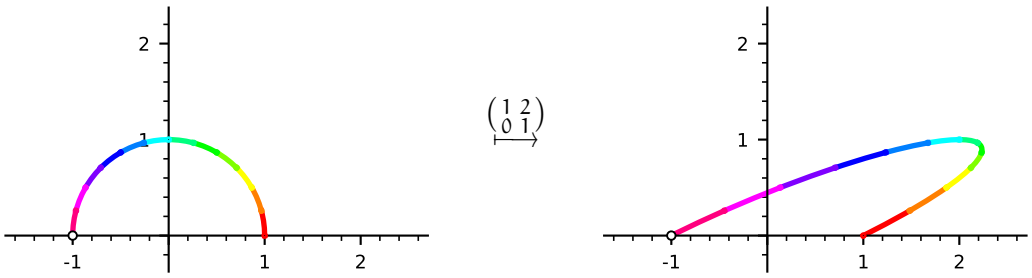

The next transformation is a shear.[1]

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

Because of the first output component's $2y$, input vectors with $y$'s of large absolute value have their first component affected more than do input vectors near the $x$-axis.

```
sage: q = plot_circle_action(1,0,0,1)
sage: q.set_axes_range(-1.5, 2.5, -.25, 1.25)
None
sage: q.save("graphics/svd004a.pdf")
None
sage: p = plot_circle_action(1,2,0,1)
sage: p.set_axes_range(-1.5, 2.5, -.25, 1.25)
None
sage: p.save("graphics/svd004b.pdf")
None
```

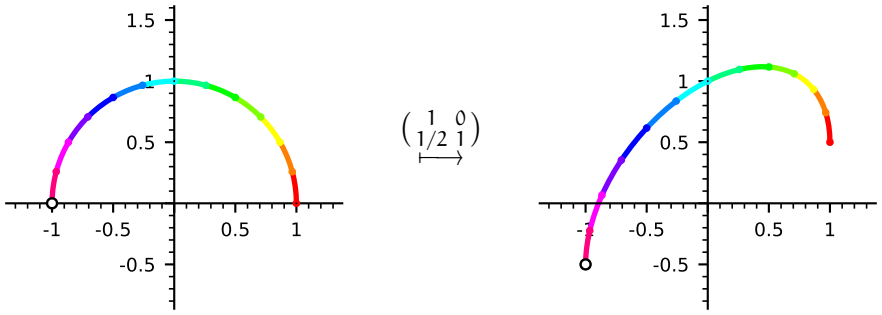
[1] A shear is the motion of two parts of a body where they slide relative to each other in a direction parallel to their plane of contact. Make a stack of newpapers and lean against them. The papers will slide parallel to each other, with the top ones sliding the furthest.




Next is a shear where it is the output's second component that is is affected, here by the input's distance from the x-axis.

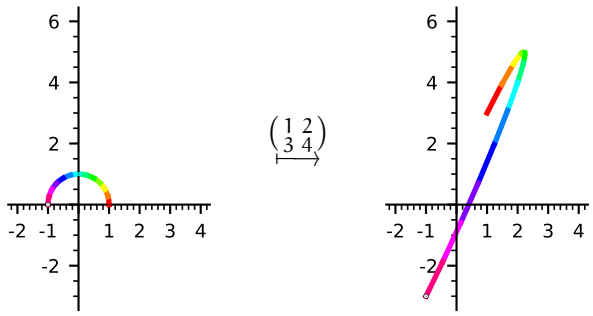
$$\begin{pmatrix} x \\ y \end{pmatrix} \mapsto \begin{pmatrix} x \\ (1/2)x + y \end{pmatrix}$$

```
sage: q = plot_circle_action(1,0,0,1)
sage: q.set_axes_range(-1.25, 1.25, -0.75, 1.5)
None
sage: q.save("graphics/svd005a.pdf")
None
sage: p = plot_circle_action(1,0,1/2,1)
sage: p.set_axes_range(-1.25, 1.25, -0.75, 1.5)
None
sage: p.save("graphics/svd005b.pdf")
None
```




And here is a generic map.

```
sage: q = plot_circle_action(1,0,0,1)
sage: q.set_axes_range(-2, 4, -3, 6)
None
sage: q.save("graphics/svd006a.pdf")
None
sage: p = plot_circle_action(1,2,3,4)
sage: p.set_axes_range(-2, 4, -3, 6)
None
sage: p.save("graphics/svd006b.pdf")
None
```

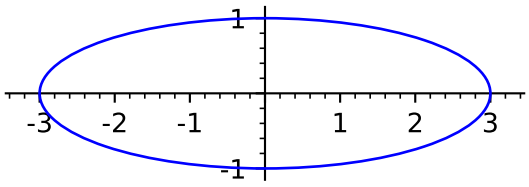



Besides rotating and shearing, it also changes orientation.

## SVD

In those pictures the image of the half circle is a half ellipse, and consequently the image of the full circle is an ellipse. Recall that in $\mathbb{R}^2$ an ellipse has a major axis, the longer one, and a minor axis.[1] Write $\sigma_1$ for the length of the semi-major axis, the distance from the center to the furthest-away point on the ellipse, and write $\sigma_2$ for the length of the semi-minor axis.

```
sage: plot.options['axes_pad'] = 0.5
sage: plot.options['fontsize'] = 4
sage: plot.options['aspect_ratio'] = 1
sage: sigma_1=3
sage: sigma_2=1
sage: E = ellipse((0,0), sigma_1, sigma_2)
sage: E.save("graphics/svd100.pdf")
None
```



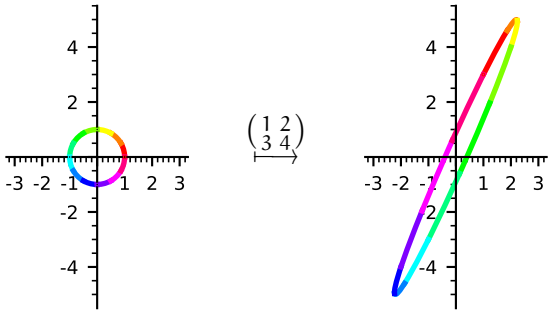

The two axes are orthogonal. Above, the major axis is on the $x$-axis while the minor is on the $y$-axis.

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 $t\colon \mathbb{R}^n \to \mathbb{R}^m$, there are bases $B = \langle \vec{\beta}_1, \ldots, \vec{\beta}_n \rangle$ for the domain and $D = \langle \vec{\delta}_1, \ldots, \vec{\delta}_m \rangle$ for the codomain such that $t(\vec{\beta}_i) = \sigma_i \vec{\delta}_i$, for $i \leq n$. The scalars $\sigma_i$ are called *singular values*. 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 $\sigma_i \vec{\delta}_i$.

[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.

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

```
sage: load("plot_action.sage")
None
sage: q = plot_circle_action(1,0,0,1,full_circle=True)
sage: q.set_axes_range(-3, 3, -5, 5)
None
sage: q.save("graphics/svd101a.pdf")
None
sage: p = plot_circle_action(1,2,3,4,full_circle=True)
sage: p.set_axes_range(-3, 3, -5, 5)
None
sage: p.save("graphics/svd101b.pdf")
None
```




(∗)

*Sage* will find the SVD of this example matrix.

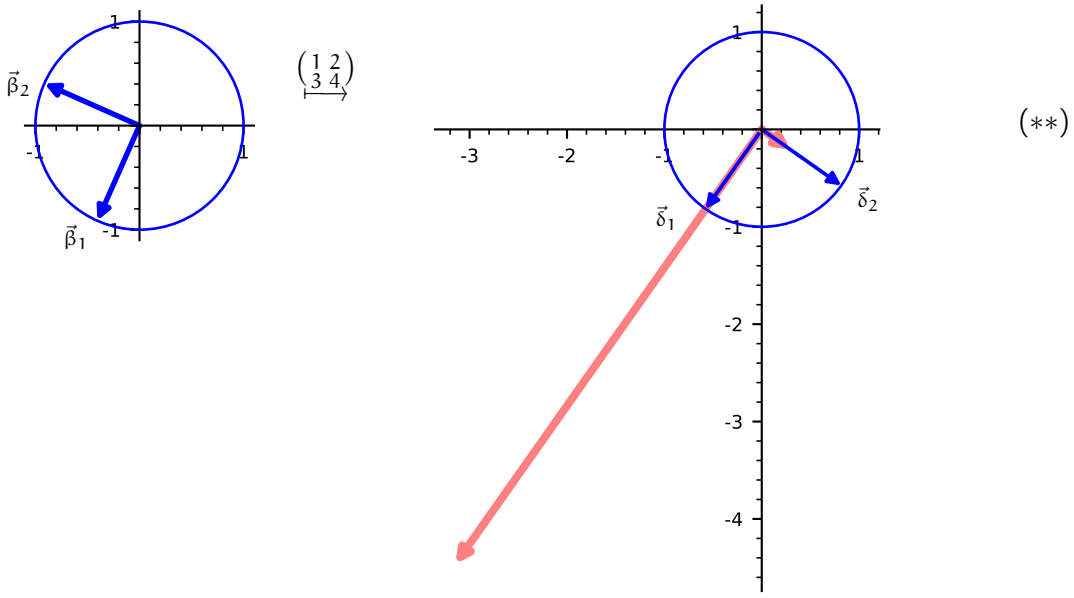
```
sage: M = matrix(RDF, [[1, 2], [3, 4]])
sage: U,Sigma,V = M.SVD()
sage: U
[-0.40455358483375703   -0.9145142956773042]
[ -0.9145142956773045    0.4045535848337568]
sage: Sigma
[ 5.464985704219043                0.0]
[               0.0  0.3659661906262574]
sage: V
[-0.5760484367663209   0.8174155604703631]
[-0.8174155604703631  -0.5760484367663209]
sage: U*Sigma*(V.transpose())
[1.0000000000000009                2.0]
[ 3.000000000000001                4.0]
```

The Singular Value Decomposition has $M$ as the product of three matrices, $U\Sigma V^{\mathrm{T}}$. 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 $\Sigma$.

We can get *Sage* to plot the effect of the transformation on the basis vectors for the domain so that we can compare those with the basis vectors for the codomain.

```
sage: M = matrix(RDF, [[1, 2], [3, 4]])
sage: U,Sigma,V = M.SVD()
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: C = circle((0,0), 1)
sage: P = C + plot(beta_1) + plot(beta_2)
sage: P.save("graphics/svd102a.pdf")
None
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)
sage: Q = Q + plot(delta_1,width=1.4)
sage: Q = Q + plot(beta_2*M,width=3)
sage: Q = Q + plot(delta_2,width=1.4)
sage: Q.save("graphics/svd102b.pdf")
None
```

Below, on the left, the domain's basis are the blue $\vec{\beta}$'s. On the right, the codomain's basis are the blue $\vec{\delta}$'s. Also on the right, in light red, are the images of the $\vec{\beta}$'s. The diagonal entries of $\Sigma$ describe the relationships: the red vector $t(\vec{\beta}_1)$ is about 5.5 times $\vec{\delta}_1$ while $t(\vec{\beta}_2)$ is about 0.4 times $\vec{\delta}_2$.



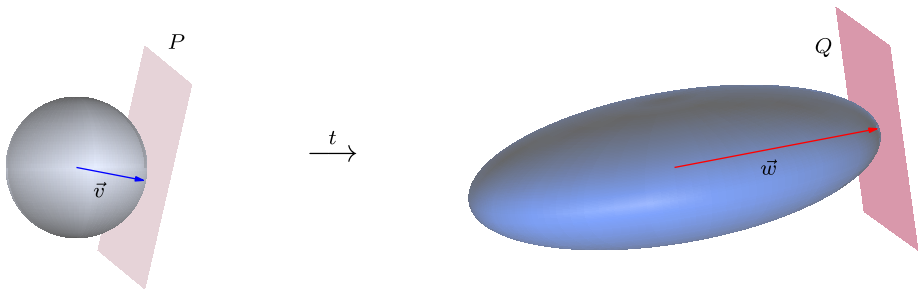

The two bases are orthonormal, comprised of unit vectors that are orthogonal. Note also, by comparing with the diagram above labeled $(*)$, that long red vector is the semi-major axis while its short red vector is the semi-minor axis.

## Proof sketch

Consider an $n \times n$ matrix T that is nonsingular. Let $t\colon \mathbb{R}^n \to \mathbb{R}^n$ be the nonsingular transformation represented by T with respect to the standard bases. We will argue that there are scalars $\sigma_0$, ... $\sigma_n$ and bases $B = \langle \vec{\beta}_1, \ldots, \vec{\beta}_n \rangle$ and $D = \langle \vec{\delta}_1, \ldots, \vec{\delta}_n \rangle$ for the domain and codomain such that $t(\vec{\beta}_i) = \sigma_i \vec{\delta}_i$.[1]

Recall Calculus I's Extreme Value Theorem: for a continuous function $f\colon \mathbb{R} \to \mathbb{R}$, if a subset $D \subset \mathbb{R}$ of the domain is closed and bounded then its image $f(D) = \{f(d) \mid d \in D\}$ is also closed and bounded (see Wikipedia [2012]). A generalization 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 a hyperellipse so we will call it that.

Because this hyperellipse is closed and bounded, it has a point furthest from the origin (if there is more than one then just pick one). Let $\vec{w}$ be the vector extending from the origin to that furthest point. The nonsingular map t is invertible so there is one and only one member of the unit sphere $\vec{v}$ such that $f(\vec{v}) = \vec{w}$. Let P be the hyperplane 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 intersects its ellipsoid only at $\vec{w}$ (if it intersected in two places then the inverse image of those two would be two places on P that intersect the sphere).




Consider sliding P along the vector $\vec{v}$, to the origin. This makes clear that P is the subspace of $\mathbb{R}^n$ consisting of vectors perpendicular to $\vec{v}$ (these elements of P have their entire bodies in P, unlike the pictured vector $\vec{v}$, which just touches P with its endpoint). This subspace has dimension $n-1$, so it is a hyperplane. We will argue in the next paragraph that similarly Q consists of vectors perpendicular to $\vec{w}$. With that, we can complete an argument by induction: we start constructing the bases B and D by taking $\vec{\beta}_1$ to be $\vec{v}$, taking $\sigma_1$ 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.

So consider Q. Since t is nonsingular, Q is an $n-1$ dimensional subspace of $\mathbb{R}^n$, a hyperplane. The hyperplane 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 hyperplane, besides P, that touches the sphere only at $\vec{v}$, which is impossible (since it is a sphere). To see that Q is the subspace perpendicular to $\vec{w}$, consider a sphere in the codomain centered at the origin whose radius is the length $|\vec{w}|$. 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, that is, the tangent plane of the ellisoid, touches that ellipsoid only at $\vec{w}$. Therefore this tangent plane is Q. That ends the argument.

[1] This argument, from Blank et al. [1989], is a sketch because it uses results that many readers have seen in a form not as strong as needed for the full proof, and because it relies on material from the book that is optional. In addition, we'll consider only the case of a nonsingular matrix. This gives the main idea, which is the point of a sketch.

## 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, $A = U\Sigma V^{\mathrm{T}}$. The $m \times n$ matrix $\Sigma$ is all zeroes except for diagonal entries, the singular values, $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r > 0$ 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, called the left and right singular vectors for $A$.

```
sage: M = matrix(RDF, [[0, 1, 2], [3, 4, 5]])
sage: U,Sigma,V = M.SVD()
sage: U
[-0.2747211278973781  -0.9615239476408235]
[-0.9615239476408234   0.2747211278973778]
sage: Sigma
[7.3484692283495345                0.0                0.0]
[              0.0 0.9999999999999994                0.0]
sage: V
[-0.3925405078644311   0.8241633836921343  0.40824829046386296]
[-0.5607721540920443    0.137360563948689 -0.8164965809277261]
[-0.7290038003196575 -0.5494422557947561   0.4082482904638631]
```

The number of singular values is the row rank of the matrix. Here *Sage* gets a $\sigma_2$ that is not quite 1 because of numerical issues.

We can simplify the product $U\Sigma V^{\mathrm{T}}$. Consider the case where all three matrices are $2 \times 2$.

$$\begin{aligned}\begin{pmatrix} u_{1,1} & u_{1,2} \\ u_{2,1} & u_{2,2} \end{pmatrix} \begin{pmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{pmatrix} \begin{pmatrix} v_{1,1} & v_{1,2} \\ v_{2,1} & v_{2,2} \end{pmatrix} &= \begin{pmatrix} u_{1,1} & u_{1,2} \\ u_{2,1} & u_{2,2} \end{pmatrix} \left[ \begin{pmatrix} \sigma_1 & 0 \\ 0 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 0 \\ 0 & \sigma_2 \end{pmatrix} \right] \begin{pmatrix} v_{1,1} & v_{1,2} \\ v_{2,1} & v_{2,2} \end{pmatrix} \\ &= \sigma_1 \begin{pmatrix} u_{1,1} & u_{1,2} \\ u_{2,1} & u_{2,2} \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} v_{1,1} & v_{1,2} \\ v_{2,1} & v_{2,2} \end{pmatrix} \\ &\quad + \sigma_2 \begin{pmatrix} u_{1,1} & u_{1,2} \\ u_{2,1} & u_{2,2} \end{pmatrix} \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} v_{1,1} & v_{1,2} \\ v_{2,1} & v_{2,2} \end{pmatrix}\end{aligned}$$

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.

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

The second term simplifies in the same way.

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

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

## Application: data compression

Suppose that a 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 $n^2$-many floating points. For instance, if $n = 500$ then it has $500^2 = 250\,000$ floats. That's a lot of data. We will show a way to reduce it.

We've just seen how to write the matrix as a sum, $M = \sigma_1 \cdot \vec{u}_1\vec{v}_1^{\mathrm{T}} + \sigma_2 \cdot \vec{u}_2\vec{v}_2^{\mathrm{T}} + \cdots$, where the vectors have unit length and the scalars decrease, $\sigma_1 \geq \sigma_2 \geq \cdots 0$. But that is not yet the savings, because each term in that sum requires $n$ floating points for $\vec{u}_i$, another $n$ for $\vec{v}_i$, and one more for $\sigma_i$. So if $n = 500$ that's $500 \cdot (2 \cdot 500 + 1) = 500\,500$ floats, about twice the size of the original matrix.

But the sum form does have an advantage, namely that the scalars decrease. If we throw out a lot of the terms and keep only a few with the largest $\sigma$'s, say the first 50 of them, then we get a big savings: $50 \cdot (2 \cdot 50 + 1) = 5\,050$ floats, which is about 2% of the storage size of the full matrix.

In short, if you have data as a matrix then you can hope to compress it by converting to the summation formula and dropping terms with small $\sigma$'s. The question, though, is whether you lose too much information.

The answer is that you can retain a lot. We will illustrate with image compression. This is *The Great Wave off Kanagawa* by Hokusai, ca 1830, Wikipedia [2019].[1] It depicts an rogue wave threatening three boats off what is today Yokohama, with Mount Fuji in the background.

The image compression code we will use is in `img_squeeze.sage`, listed at the end of the chapter. It breaks the picture into three matrices, for the red data, the green data, and the blue data. We want to see how badly the image degrades for various cutoffs. The code below sets the cutoff at retaining only the terms with the top 10% of singular values. For this image, that is 92 singular

[1] The image file is generously made freely available by The Art Institute of Chicago.

values retained. The code gives us some insight into their size by printing the ones for the red matrix (but we've edited out most of them).

```
sage: load("img_squeeze.sage")
sage: img_squeeze("pix/greatwave.png", "pix/greatwave_squeezed.png", 0.10)
image has 1497 rows and 922 columns
sigma_RD 0 =192773.40
sigma_RD 1 =26258.04
sigma_RD 2 =20055.53
sigma_RD 3 =16881.59
sigma_RD 4 =13394.80
sigma_RD 5 =12875.31
sigma_RD 6 =11064.42
sigma_RD 7 =10250.95
    :
sigma_RD 92 =2240.98
```

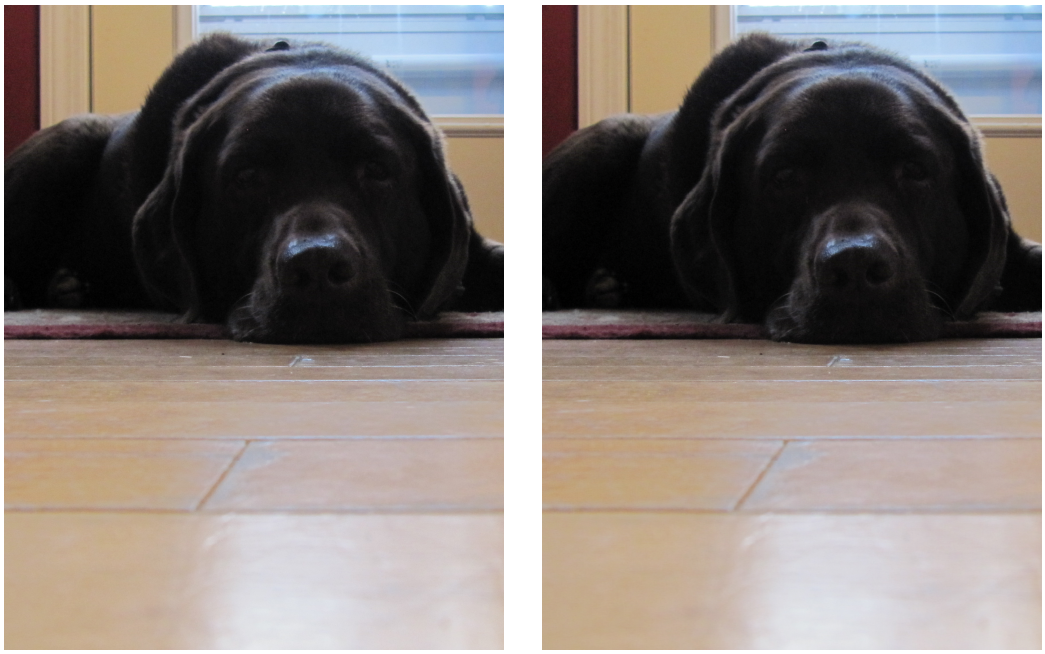
(For contrast, the eight smallest singular values are 11.22, 10.80, 9.00, 8.26, 8.08, 7.29, 6.55, and 5.88.) Below is the compressed result.

This picture shows loss. The colors are not as good, the edges are not sharp, and there are artifacts, including vertical lines extending from the writing in the upper left. Basically, Fiji has gotten fuzzy. But certainly the image is entirely recognizable — not bad for omiting 90 percent of the summands.

Every additional term in the summation adds to the storage and transmission requirements by about $2n$ floats. The above cutoff of 0.10 needs only two percent of the storage and transmission requirements of the original image's full matrix, although 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.

Setting the cutoff parameter to 0.20 can 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 (both are shown at 40% of their full size).



(The top singular value is 291533.51. The twenty percent cutoff is 129.23. The smallest of the 1739 singular values is 3.36.) The picture on the left is 4 megabytes, while the right is 3.4.

## Source of plot_action.sage

The `plot_circle_action` routine takes the four entries of the $2 \times 2$ matrix and returns a list of two graphics, showing the input and the output. Other parameters are the number of colors and a flag giving whether to plot a full circle or just the top half circle.

The driver routine computes the list of colors and calls a helper `color_circle_list`, given below, which returns a list of graphics. Finally, the routine plots those graphics (the `+=` adds the picture on the left to the one on the right).

```
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 routine does the heavy lifting. There are two globals. The variable `CIRCLE_THICKNESS` sets the thickness of the pen that draws the plotted curve, in points (a printer's unit, which are $1/72$ inch). Similarly `DOT_SIZE` sets the size of the small empty circle (also in inches). This routine produces a parametrized curve $(x(t), y(t))$ and uses *Sage*'s `parametric_plot` function to yield 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
    """
    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)+b*sin(t)
        y(t) = c*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 the small empty circle at the end to show that the image of $(-1, 0)$ is not part of the graph.

## Source of img_squeeze.sage

We use the Python Image Library for reading and writing the graphic.

```
from PIL import Image
```

The function `img_squeeze` takes three arguments: the names of the two files, and the cutoff real number between 0 and 1 that gives the percentage of the singular values to retain in the sum.

```
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 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
    dim_bound = min(rows, cols)  # for non-square images
    cutoff = int(round(percent*dim_bound,0))
    print("image has", rows, "rows and", cols, "columns")
    # 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 look 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
    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, "=%0.2f" % Sigma_RD[i][i])
    print("    :")  # vdots
```

```
print("sigma_RD",cutoff,"=%0.2f" % Sigma_RD[cutoff][cutoff])
print("    :")  # vdots
for i in range(dim_bound-8, dim_bound):
    print(" at bottom: sigma_RD", i, "=%0.2f" % Sigma_RD[i][i])
```

Finally, for each matrix we compute the sum $\sigma_1 \cdot \vec{u}_1\vec{v}_1^{\mathrm{T}} + \sigma_2 \cdot \vec{u}_2\vec{v}_2^{\mathrm{T}} + \cdots$, 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).column())
    v_i = matrix(RDF, V_RD.column(i).column().transpose())
    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).column())
    v_i = matrix(RDF, V_GR.column(i).column().transpose())
    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).column())
    v_i = matrix(RDF, V_BL.column(i).column().transpose())
    A_BL = copy(A_BL) + sigma_i*u_i*v_i
```

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):
    for col in range(cols):
        p = (int(A_RD[row][col]),
             int(A_GR[row][col]),
             int(A_BL[row][col]))
        img_squoze.putpixel((row,col), p)
img_squoze.save(fn_out)
```

(This part of the routine takes a long time, a number of seconds, in part because the code is intended to be easy to read rather than fast to run.)

# Geometry of Linear Maps

To show the geometric effect of linear transformations, we will picture the actions of plane transformations on the unit square. We show the plane, $\mathbb{R}^2$, because it fits on the paper, but the principles extend to higher-dimensional spaces $\mathbb{R}^n$.

## Lines map to lines

The prior chapter points out that the condition $h(r\cdot\vec{v})=r\cdot h(\vec{v})$ in the definition of linear map means that these maps send lines through the origin to lines through the origin. What about lines not through the origin? Fix a linear map $h\colon\mathbb{R}^d\to\mathbb{R}^c$. A line in the domain has the form $\ell=\{\vec{m}\cdot s+\vec{b}\mid s\in\mathbb{R}\}$ for $\vec{m},\vec{b}\in\mathbb{R}^d$ (if $\vec{m}=\vec{0}$ then the line is degenerate). Its image

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

is a line in the codomain. So a linear map sends any line, through the origin or not, to another line.

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

$$\mathrm{Rep}_{\mathcal{E}_2,\mathcal{E}_2}(t)=\begin{pmatrix}\cos(\pi/6)&-\sin(\pi/6)\\ \sin(\pi/6)&\cos(\pi/6)\end{pmatrix}=\begin{pmatrix}\sqrt{3}/2&-1/2\\ 1/2&\sqrt{3}/2\end{pmatrix}$$

And consider the line $y=3x+2$.

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

Under the action of $t$,

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

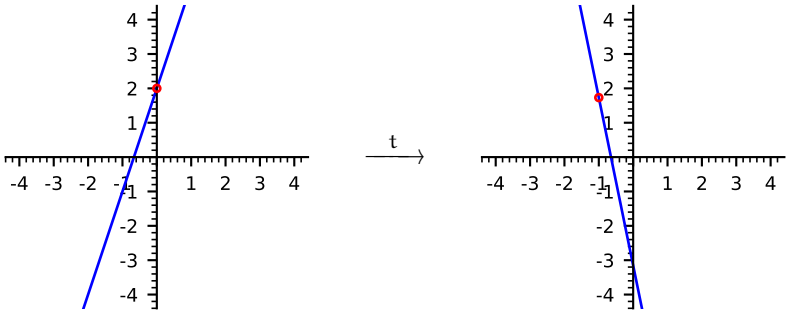
that line becomes this set.

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

Here is the code and the input and output plots.[1]

[1] As in earlier chapters, we've omitted some of the code that sets the font size, etc.

```
sage: s = var('s')
sage: ell = parametric_plot((s, 3*s+2), (s, -10, 10))
sage: d = circle((0,2), 0.1, rgbcolor=(1,0,0))
sage: ell = ell+d
sage: ell.save("graphics/geo000a.pdf")
None
sage: t_x(s) = ((sqrt(3)-3)/2)*s-1
sage: t_y(s) = ((1+3*sqrt(3))/2)*s+sqrt(3)
sage: t_ell = parametric_plot((t_x(s), t_y(s)), (s, -10, 10))
sage: d = circle((-1,sqrt(3)), 0.1, rgbcolor=(1,0,0))
sage: t_ell = t_ell+d
sage: t_ell.save("graphics/geo000b.pdf")
None
```



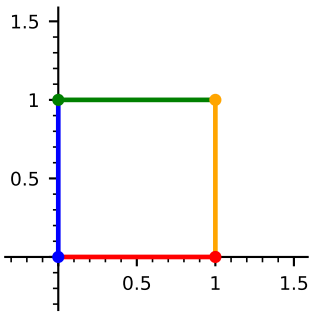

The map t is rotation by $\pi/6$ radians, but that can be hard to see. One example, shown with red dots, is that the vector that ends at $(0,2)$ is rotated to the vector that ends at $(-1,\sqrt{3})$.

## The unit square

The prior chapter illustrated the effect of plane transformations $t\colon \mathbb{R}^2 \to \mathbb{R}^2$ by applying them to the unit circle. In this chapter we use the unit square. The observation that linear maps send lines to lines makes the code for this easy: we start with the four corners of the input square, compute the effect of the transformation on those to make four output corners, and then connect those four with line segments, getting a parallelogram. (The equations above show that lines with the same direction vector $\vec{m}$ map to lines with the same direction vector $t(\vec{m})$, that is, parallel lines map to parallel lines.)

So, our method will be to start with the unit square. Note the colors; they will play the same role here as they did earlier.

```
sage: load("plot_action.sage")
None
sage: p = plot_square_action(1,0,0,1)  # identity matrix
sage: p.save("graphics/geo100.pdf")
None
```

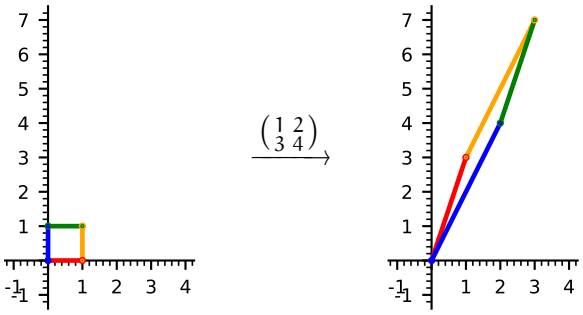



The `plot_square_action(a,b,c,d)` routine applies to that unit square the transformation represented with respect to the standard basis by the matrix with entries $a$, $b$, $c$, and $d$. (The source code is at the end of the chapter.)

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} ax+by \\ cx+dy \end{pmatrix}$$

This pictures of the effect of the generic matrix.

```
sage: load("plot_action.sage")
None
sage: q = plot_square_action(1,0,0,1)
sage: q.set_axes_range(-1, 4, -1, 7)
None
sage: q.save("graphics/geo101a.pdf")
None
sage: p = plot_square_action(1,2,3,4)
sage: p.set_axes_range(-1, 4, -1, 7)
None
sage: p.save("graphics/geo101b.pdf")
None
```

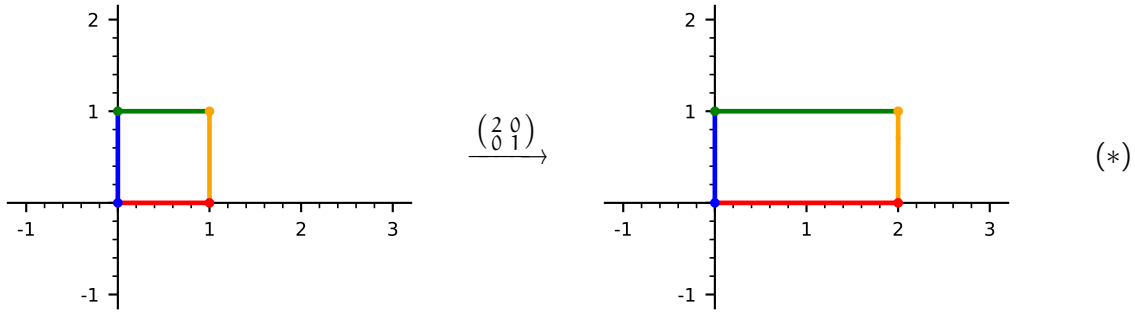



As described above, it transforms the input square into an output parallelogram.

The colors show another effect of the transformations, beyond shape-changing. Taking the colors in their natural order of red, orange, green, and blue, the domain square has a counterclockwise orientation. But the codomain's figure is clockwise. So the colors illustrate that this transformation reverses orientation (sometimes called 'sense').

We can hope to understand complex behavior by building up from an understanding of simple behavior. So we start by looking at the same example transformation as in the prior chapter, the one that doubles the first component.

```
sage: load("plot_action.sage")
None
sage: q = plot_square_action(1,0,0,1)
sage: q.set_axes_range(-1, 3, -1, 2)
None
sage: q.save("graphics/geo102a.pdf")
None
sage: p = plot_square_action(2,0,0,1)
sage: p.set_axes_range(-1, 3, -1, 2)
None
sage: p.save("graphics/geo102b.pdf")
None
```

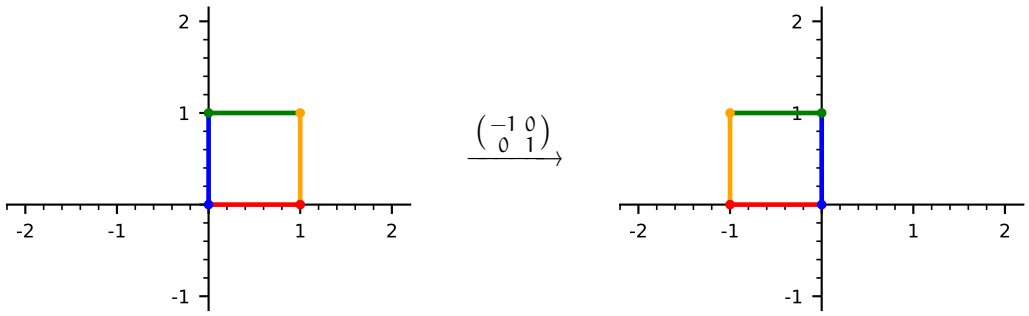



(∗)

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 if you take $-1$ times the $x$-coordinate?

```
sage: load("plot_action.sage")
None
sage: q = plot_square_action(1,0,0,1)
sage: q.set_axes_range(-2, 2, -1, 2)
None
sage: q.save("graphics/geo103a.pdf")
None
sage: p = plot_square_action(-1,0,0,1)
sage: p.set_axes_range(-2, 2, -1, 2)
None
sage: p.save("graphics/geo103b.pdf")
None
```



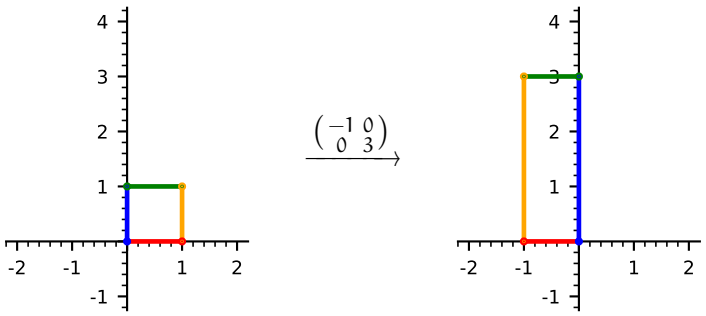

(**)

It changes the orientation.

We say that the shape on the left above has an *oriented area* of $-1$. The motivation for taking the area with a sign is continuity: imagine starting with the right-hand figure from the example before this one, the diagram labeled $(*)$, and sliding 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 measure from the grade school meaning of area. The grade school version 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: load("plot_action.sage")
None
sage: q = plot_square_action(1,0,0,1)
sage: q.set_axes_range(-2, 2, -1, 4)
None
sage: q.save("graphics/geo104a.pdf")
None
sage: p = plot_square_action(-1,0,0,3)
sage: p.set_axes_range(-2, 2, -1, 4)
None
sage: p.save("graphics/geo104b.pdf")
None
```

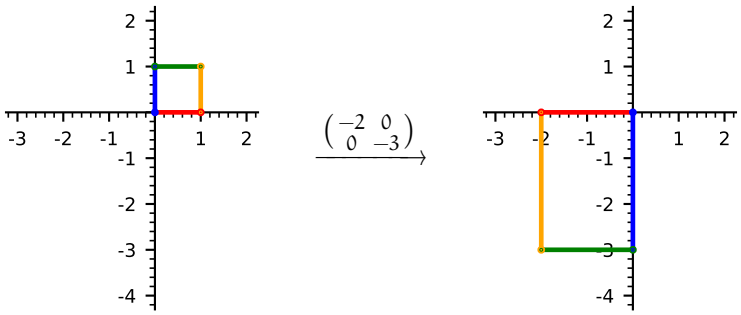



The colors show that this transformation also changes the orientation, so the new shape has an oriented area of $-3$.

What if we change the orientation twice?

```
sage: load("plot_action.sage")
None
sage: q = plot_square_action(1,0,0,1)
sage: q.set_axes_range(-3, 2, -4, 2)
None
sage: q.save("graphics/geo105a.pdf")
None
sage: p = plot_square_action(-2,0,0,-3)
sage: p.set_axes_range(-3, 2, -4, 2)
None
sage: p.save("graphics/geo105b.pdf")
None
```

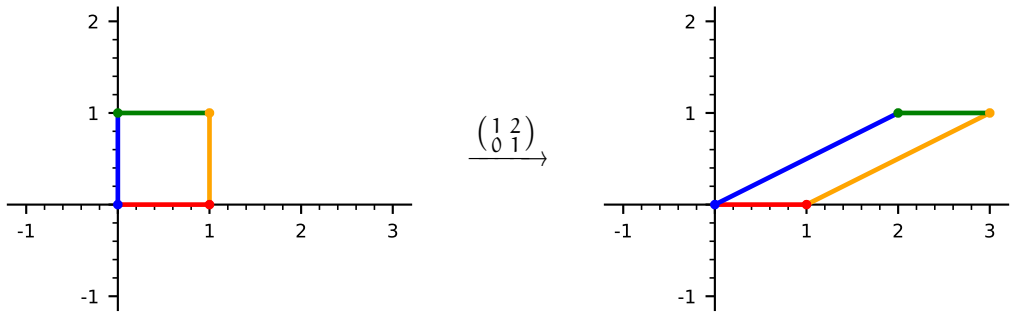
The before and after plots




show that on the right the colors come in the same order as they did in the original shape's counterclockwise red, orange, green, and then blue. That is, reversing orientation twice brings you back to the original orientation. The new shape has an oriented area of 6.

Next consider the effect of putting off-diagonal entries in the matrix.

```
sage: load("plot_action.sage")
None
sage: q = plot_square_action(1,0,0,1)
sage: q.set_axes_range(-1, 3, -1, 2)
None
sage: q.save("graphics/geo106a.pdf")
None
sage: p = plot_square_action(1,0,2,1)
sage: p.set_axes_range(-1, 3, -1, 2)
None
sage: p.save("graphics/geo106b.pdf")
None
```



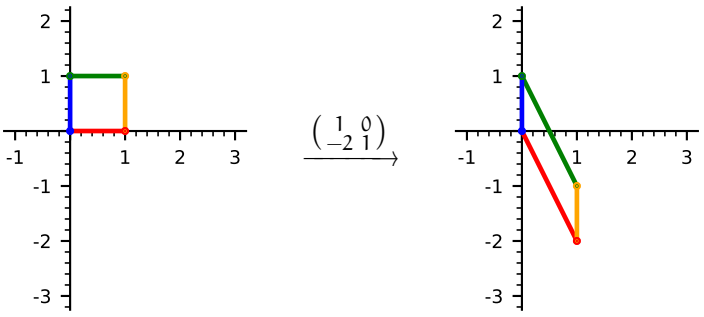

This transformation is a *shear*. The output sides are not at right angles, although still opposite sides are parallel. The action

$$\begin{pmatrix} 1 & 2 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x + 2y \\ y \end{pmatrix}$$

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, etc. That is, vectors are shifted depending on how far they are above or below the $x$-axis. The output shape has a base of 1 with a height of 1, and the orientation is preserved, so its oriented area is 1.

Putting a nonzero value in the other off-diagonal entry of the matrix, the lower left, has the same effect except that it shears parallel to the $y$-axis.

```
sage: load("plot_action.sage")
None
sage: q = plot_square_action(1,0,0,1)
sage: q.set_axes_range(-1, 3, -3, 2)
None
sage: q.save("graphics/geo107a.pdf")
None
sage: p = plot_square_action(1,0,-2,1)
sage: p.set_axes_range(-1, 3, -3, 2)
None
sage: p.save("graphics/geo107b.pdf")
None
```




The action above

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

means that vectors are shifted depending on how far they are from the $x$ axis. For instance, an input vector with an $x$ component of 1 is shifted by $-2$ while if the $x$ component is 2 then it is shifted by $-4$. 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 diagram $(*)$ above, the matrix has determinant 2 and it doubles the oriented area. In diagram $(**)$, the matrix multiplies the oriented area by $-1$.

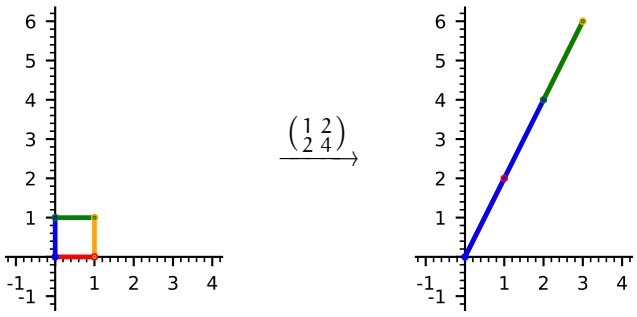
One advantage of starting these before/after pictures with a unit square is that because the input has area 1, as a result the oriented area of the output shape equals the determinant of the matrix.

For instance, the matrix

$$\begin{pmatrix} 1 & 2 \\ 2 & 4 \end{pmatrix}$$

is singular so it has determinant 0.

```
sage: load("plot_action.sage")
None
sage: q = plot_square_action(1,0,0,1)
sage: q.set_axes_range(-1, 4, -1, 6)
None
sage: q.save("graphics/geo200a.pdf")
None
sage: p = plot_square_action(1,2,2,4)
sage: p.set_axes_range(-1, 4, -1, 4)
None
sage: p.save("graphics/geo200b.pdf")
None
```




The output shape has an area of zero.

## 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 we 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 $2\rho_1 + \rho_2$.

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

As described in the book, the elementary reduction matrices come in the three types. For a matrix H, do row scaling $r\rho_i$ with $M_i(r)\,H$, swap rows $\rho_i \leftrightarrow \rho_j$ with $P_{i,j}\,H$, and add a multiple of one row to another $k\rho_i + \rho_j$ with $C_{i,j}(k)\,H$. These three arise from applying a row operation to an identity matrix:

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

(where $r \neq 0$ and $i \neq j$). For instance, the prior paragraph used the $3\times3$ matrix $C_{1,2}(2)$. We will focus on transformations, so we will take all of these matrices to be the same-size and square.

Continuing the Gauss's Method reduction started in the matrix equation above, use $C_{2,3}(1)$ to perform $\rho_2 + \rho_3$, producing echelon form.

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

Observe 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 $j > i$. The elementary matrices that perform those operations are called *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*.)

We can go further. Next use a diagonal matrix to make the leading entries of the nonzero rows of the echelon form matrix into 1's.

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

Finally, use column operations to go all the way to a block partial identity matrix. Here is right-multiplication on the right-hand side of $(*)$ to add $-1/3$ times the first column to the second column.

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

Finally, adding $-4/3$ times the first column to the third column leaves an identity matrix.

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

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

$$D_0 \cdot L_s \cdots L_2 L_1 \cdot A \cdot U_1 U_2 \cdots U_r = I$$

Here $I$ is a partial identity matrix, $D_0$ is a diagonal 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. That means we can do the algebra to move everything but $A$ to the left side of the equation. Therefore, if you don't need any swaps in a Gauss-Jordan reduction of a matrix $A$ then you get a factorization of the starting matrix

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

where $D$ is the product of $D_0^{-1}$ and the partial identity $I$, which is clearly a diagonal matrix.

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

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

Now for the coup de gras. The inverse of a lower triangular matrix is a lower triangular matrix and the product of lower triangular matrices is lower triangular, so you can combine all the $L_i^{-1}$'s in $(***)$ into one lower triangular $L$. Likewise, you can combine all the $U_i^{-1}$'s into a single upper triangular $U$. Thus, any matrix factors 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 write $\chi_i$ for the columns.) This is the associated factorization.

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

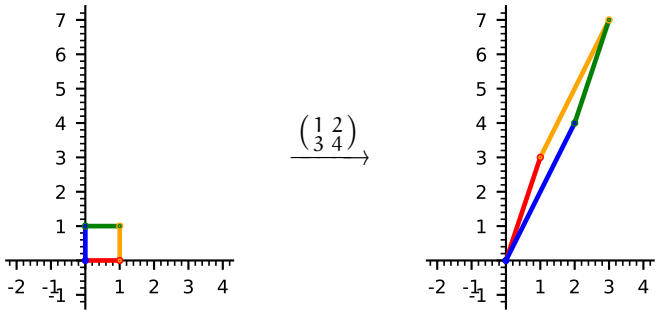
The product checks out.

```
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.

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

```
sage: load("plot_action.sage")
None
sage: q = plot_square_action(1,0,0,1)
sage: q.set_axes_range(-2, 4, -1, 7)
None
sage: q.save("graphics/geo300a.pdf")
None
sage: p = plot_square_action(1,2,3,4)
sage: p.set_axes_range(-2, 4, -1, 7)
None
sage: p.save("graphics/geo300b.pdf")
None
```




Expand it using the above LDU factorization.

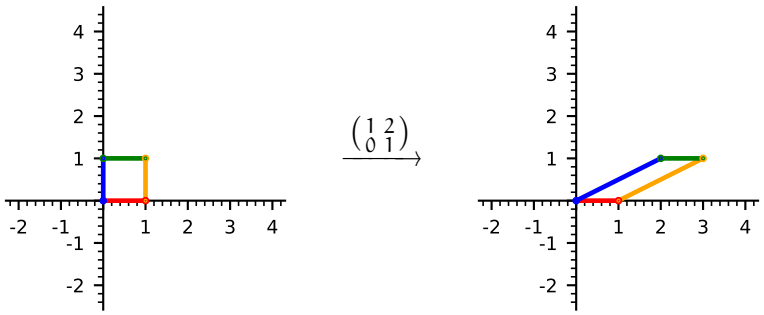
$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 3 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & -2 \end{pmatrix} \begin{pmatrix} 1 & 2 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

We will first look at the effect of each of U, D, and L separately. We will then look at their cumulative effect: the action of U, then of DU, and finally of LDU.

The matrix applied first, the rightmost matrix U, is a skew parallel to the x-axis.

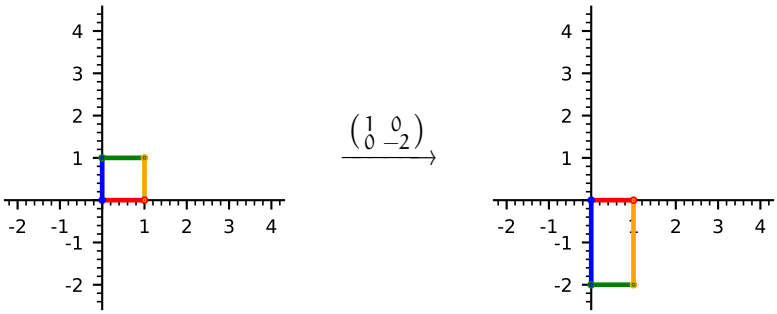
```
sage: load("plot_action.sage")
None
sage: q = plot_square_action(1,0,0,1)
sage: q.set_axes_range(-2, 4, -2.25, 4.25)
None
sage: q.save("graphics/geo303a.pdf")
None
sage: p = plot_square_action(1,2,0,1)
sage: p.set_axes_range(-2, 4, -2.25, 4.25)
None
```

```
sage: p.save("graphics/geo303b.pdf")
None
```




The second matrix D rescales and changes the orientation.

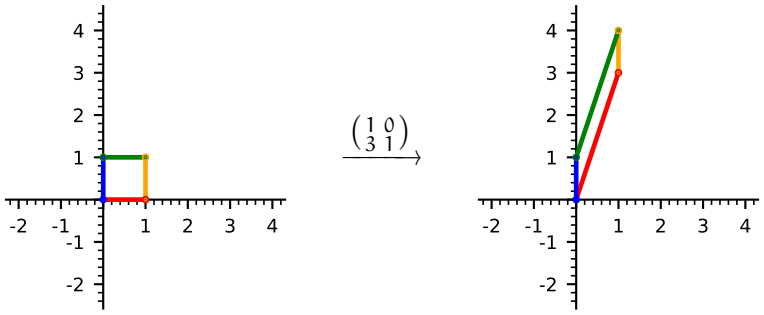
```
sage: load("plot_action.sage")
None
sage: q = plot_square_action(1,0,0,1)
sage: q.set_axes_range(-2, 4, -2.25, 4.25)
None
sage: q.save("graphics/geo302a.pdf")
None
sage: p = plot_square_action(1,0,0,-2)
sage: p.set_axes_range(-2, 4, -2.25, 4.25)
None
sage: p.save("graphics/geo302b.pdf")
None
```




The leftmost matrix, L, is a skew parallel to the y-axis.

```
sage: q = plot_square_action(1,0,0,1)
sage: q.set_axes_range(-2, 4, -2.25, 4.25)
None
sage: q.save("graphics/geo301a.pdf")
None
sage: p = plot_square_action(1,0,3,1)
sage: p.set_axes_range(-2, 4, -2.25, 4.25)
None
```

```
sage: p.save("graphics/geo301b.pdf")
None
```




The lower-triangular and upper-triangular matrices do not change orientation. Any orientation changing in LDU happens via diagonal entries that are negative. (Note that 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.)

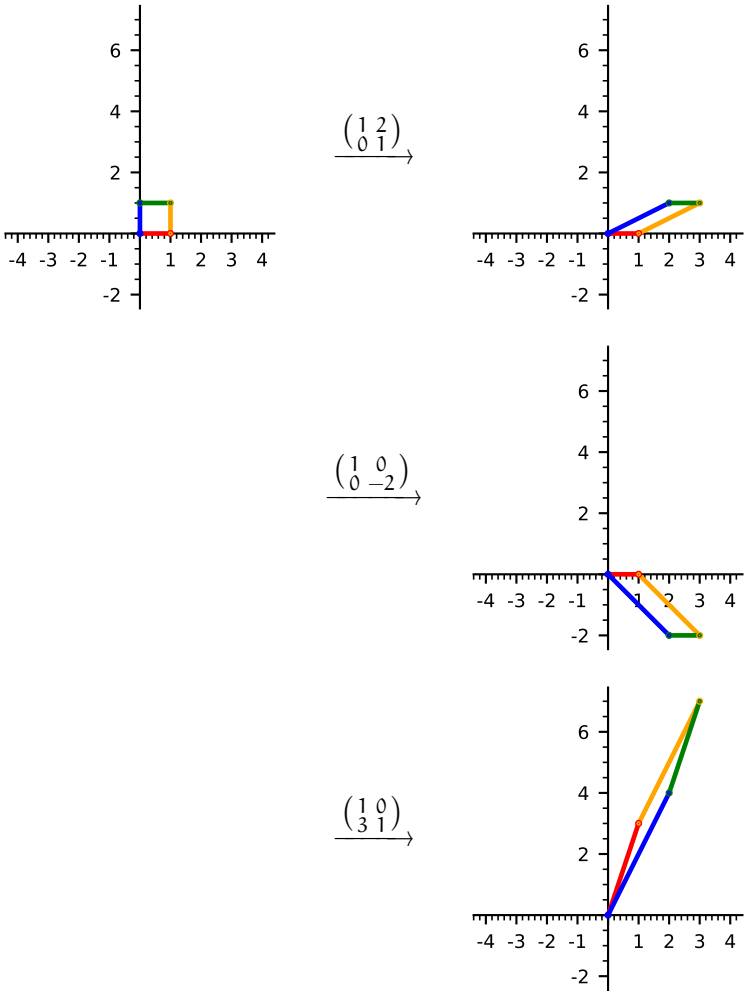
We finish our study of LDU by considering the cumulative effect. Consider in order the actions of U, then DU, and then LDU. For example, here is the cumulative effect of the maps on the unit square's upper right corner.

$$\begin{pmatrix}1\\1\end{pmatrix} \overset{U}{\longmapsto} \begin{pmatrix}3\\1\end{pmatrix} \overset{D}{\longmapsto} \begin{pmatrix}3\\-2\end{pmatrix} \overset{L}{\longmapsto} \begin{pmatrix}3\\7\end{pmatrix}$$

Here are the drawings for the entire square. Again we see a skew parallel to the $x$-axis, followed by a rescaling and change of orientation, followed by a skew parallel to the $y$-axis.

```
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: DU = D*U
sage: LDU = L*DU
sage: load("plot_action.sage")
None
sage: p = plot_square_action(1,0,0,1)
sage: p.set_axes_range(-4, 4, -2, 7)
None
sage: p.save("graphics/geo304a.pdf")
None
sage: p = plot_square_action(U[0][0], U[0][1], U[1][0], U[1][1])
sage: p.set_axes_range(-4, 4, -2, 7)
None
sage: p.save("graphics/geo304b.pdf")
None
sage: p = plot_square_action(DU[0][0], DU[0][1], DU[1][0], DU[1][1])
```

```
sage: p.set_axes_range(-4, 4, -2, 7)
None
sage: p.save("graphics/geo304c.pdf")
None
sage: p = plot_square_action(LDU[0][0], LDU[0][1], LDU[1][0],LDU[1][1])
sage: p.set_axes_range(-4, 4, -2, 7)
None
sage: p.save("graphics/geo304d.pdf")
None
```




## Source of plot_action.sage

The `plot_square_action` routine takes the four entries of the $2 \times 2$ matrix and returns a list of graphics.

```
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
```

Most of the work is done by the helper `color_square_list`.

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

There are two technical points that bear explanation. The `ZORDER` 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. Also, the way line segments butt against each other is ugly so we cover that with a dot.

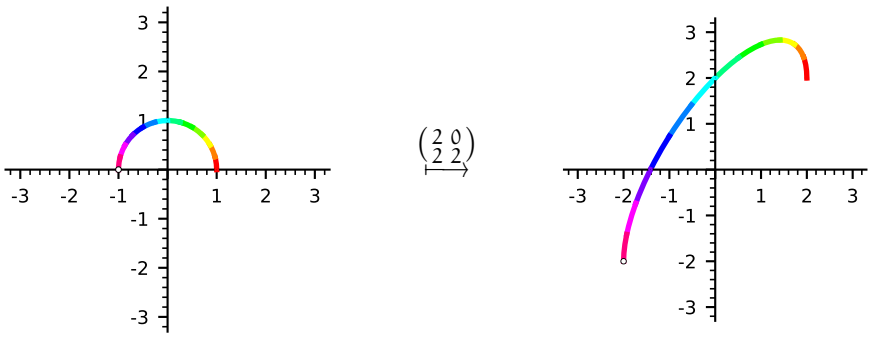
# Eigenvalues

In the chapter on Singular Value Decomposition, we considered the factor by which transformations resize vectors and found the maximum resizing factors, the singular values. In this chapter we return to plotting the image of the unit circle to consider another geometric effect, the amount by which vectors are rotated.

## Turning

Recall that a linear map has the same effect on all vectors that lie on the same line through the origin. So we can see its action by plotting what it does to one point on each such line, using the routine `plot_circle_action`.[1]

```
sage: load("plot_action.sage")
None
sage: q = plot_circle_action(1,0,0,1)
sage: q.save("graphics/eigen000a.pdf")
None
sage: p = plot_circle_action(2,0,2,2)
sage: p.save("graphics/eigen000b.pdf")
None
```



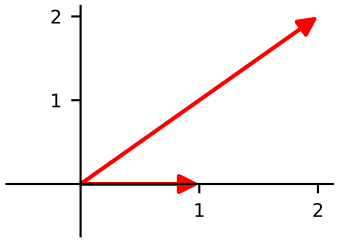

On the left, proceeding counterclockwise, the curve begins with red at $(x, y) = (1, 0)$. On the right plot it begins with red at $(1, 2)$. Here are the before and after vectors for that red point.

[1] As in earlier chapters, these plots are drawn using some options that are not shown. For the full list, see this manual's source.

```
sage: load("plot_action.sage")
None
sage: p = plot_before_after_action(2,0,2,2, [(1,0)], ['red'])
sage: p.set_axes_range(-0.5, 2, -0.5, 2)
None
sage: p.save("graphics/eigen001.pdf", ticks=([1,2],[1,2]))
None
```

The action of the matrix

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



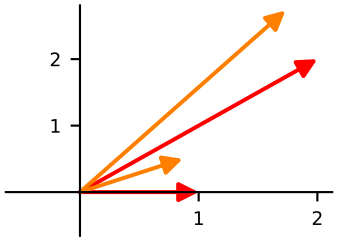

is to both resize and rotate.

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

However, with other vectors this map will resize and rotate by other amounts. That is, the action is not uniform; as we move across the upper half circle from $(1,0)$ to $(-1,0)$ the effect of the map varies. Here is the effect of the transformation on the point $(\cos(\pi/6), \sin(\pi/6))$.

```
sage: v = vector(RR, [cos(pi/6),sin(pi/6)])
sage: w = M*v
sage: v.norm(), w.norm()
(1.00000000000000, 3.23482636553150)
sage: angle_in_rads = arccos(w*v/(w.norm()*v.norm()))
sage: angle_in_rads
0.482170608511459
sage: load("plot_action.sage")
None
sage: p = plot_before_after_action(2,0,2,2, [(1,0), (cos(pi/6),sin(pi/6))], rainbow(12))
sage: p.set_axes_range(-0.5, 2, -0.5, 2.65)
None
sage: p.save("graphics/eigen001a.pdf", ticks=([1,2],[1,2]))
None
```

This shows again the before-and-after on the red input $(1,0)$, and then overlays the same effect on the orange input $(\cos(\pi/6),\sin(\pi/6))$.

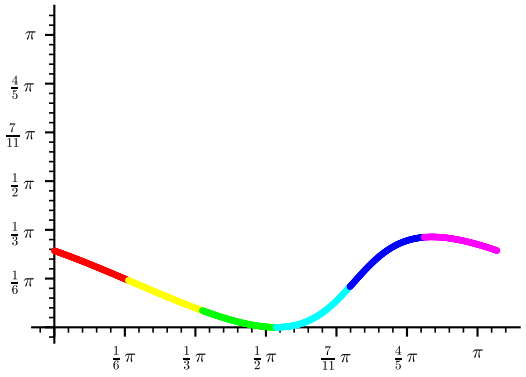



The orange output is not rotated from the input as much (it is also slightly longer).

*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 inputs the entries of a transformation and applies that map to vectors in the half circle, computes the angles by which those vectors are rotated, and draws a graph. That graph colors the output vectors, making a visual match.

```
sage: p = plot_color_angles(2,0,2,2)
sage: p.set_axes_range(0,pi,0,pi)
None
sage: p.save("graphics/eigen003.pdf")
None
```

The transformation for this graph is the map represented by $\left(\begin{smallmatrix}2&0\\2&2\end{smallmatrix}\right)$ with respect to the standard basis.

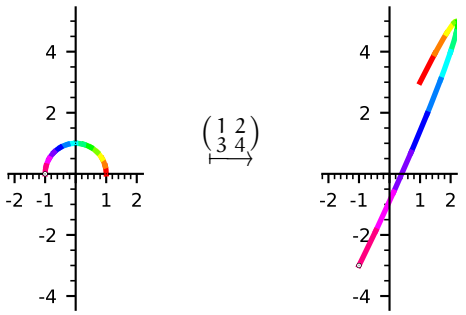



Its most interesting point is where the output angle is 0, where the graph hits the horizontal axis This happens at the input angle $\pi/2$. This is a input vector that the transformation does not turn at all — it is resized but not rotated. On the input/output diagram on page 87, it is the green input vector lying on the $y$-axis, whose associated green output also lies on the $y$-axis.

**Generic matrix** We can do the same analysis for our usual generic $2\times 2$ matrix.

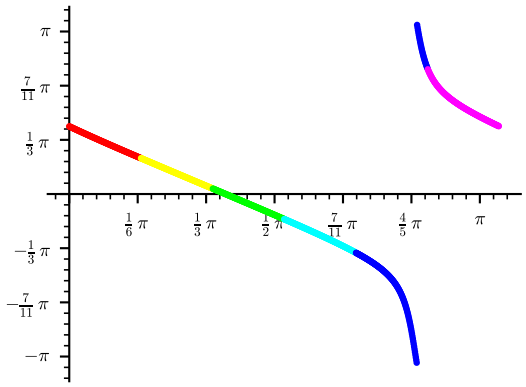
```
sage: q = plot_circle_action(1,0,0,1)
sage: q.set_axes_range(-2, 2, -4, 5)
None
```

```
sage: q.save("graphics/eigen100a.pdf")
None
sage: p = plot_circle_action(1,2,3,4)
sage: p.set_axes_range(-2, 2, -4, 5)
None
sage: p.save("graphics/eigen100b.pdf")
None
```




This graphs the angle between the each input vector and its associated output vector.

```
sage: p = plot_color_angles(1,2,3,4)
sage: p.set_axes_range(0,pi,-1*pi,pi)
None
sage: p.save("graphics/eigen101.pdf", tick_formatter=[pi,pi])
None
```




(*)

This graph has two interesting points: where $y = 0$, and where $y = \pi$ (and $y = -\pi$). In both places, the vector is resized but not turned. In the second case, where the vector is turned by an angle of $\pi$ (or $-\pi$), the input vector is on the same line through the origin as the output but the transformation rescales it by a negative number.

A vectors that is purely resized by a transformation $t$ is an *eigenvector* for $t$ (we disallow the zero vector as an eigenvector, as it is trivially resized by every transformation). The amount by which the eigenvector is resized is the associated *eigenvalue*. For an eigenvalue $\lambda$, the set of vectors associated with that rescaling factor is an *eigenspace*. *Sage* will compute the eigenspaces

of a matrix.[1]

```
sage: M = matrix(RDF, [[1, 2], [3, 4]])
sage: evs = M.eigenvectors_right()
sage: evs
[(-0.3722813232690143, [(-0.8245648401323938, 0.5657674649689923)], 1), (5.372281323269014,
    [(-0.4159735579192842, -0.9093767091321241)], 1)]
sage: evs[0]
(-0.3722813232690143, [(-0.8245648401323938, 0.5657674649689923)], 1)
sage: v0 = vector(RDF, evs[0][1][0])
sage: v0.norm()
1.0
sage: evs[1]
(5.372281323269014, [(-0.4159735579192842, -0.9093767091321241)], 1)
sage: v1 = vector(RDF, evs[1][1][0])
sage: v1.norm()
1.0
```

That shows a list with two elements, one for each eigenvalue. The first, `evs[0]`, has to do with the eigenvalue $\lambda_1 \approx -0.37$ and says that the associated eigenspace has a basis consisting of the single unit vector with endpoint approximately $(-0.82, 0.57)$.[2] The second element, `evs[1]`, is for $\lambda_2 \approx 5.37$ and says that its eigenspace has a basis consisting of the single vector with endpoint around $(-0.42, -0.91)$.

*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_right()
sage: v = vector(RDF, evs[0][1][0])
sage: angle_v = atan2(v[1], v[0])
sage: (angle_v/pi).n(digits=4)
0.8086
```

(Remember that the `n()` function gives the numerical value of the argument.) So the angle is something like $(4/5)\pi$, and thus 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 $\pi$. 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.

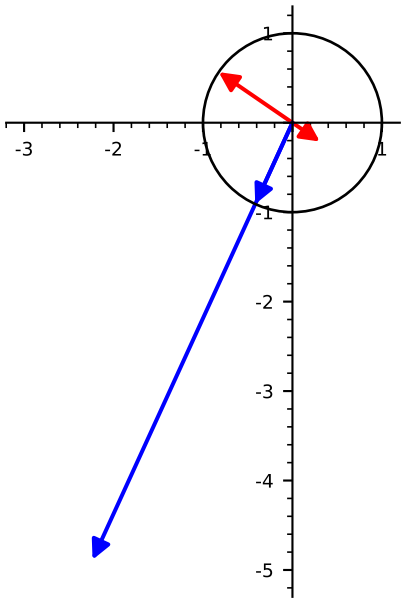
*Sage* can draw before and after pictures for the two eigenvectors.

```
sage: M = matrix(RDF, [[1, 2], [3, 4]])
sage: evs = M.eigenvectors_right()
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'])
```

[1] Whether we are taking multiplication by vectors to come from the right, $M\vec{v}$, or from the left, $\vec{v}M$, a matrix has the same eigenvalues. But the eigenvectors may be different. The *Sage* operation `eigenvectors_right` covers the book's standard $M\vec{v} = \lambda\vec{v}$ case and `eigenvectors_left` covers the other. [2] The trailing 1 is the *geometric multiplicity* of this eigenvalue, the dimension of the nullspace of $t - \lambda_1 \cdot \mathrm{id}$. We won't use this information.

```
sage: p = p1+p2+circle((0,0), 1, edgecolor='black')
sage: p.set_axes_range(-3, 1, -5, 1)
None
sage: p.save("graphics/eigen102.pdf")
None
```

This picture has two pairs of before and after vectors, one in blue and the other in red (each before vector is a unit vector). 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 $\lambda_1 \approx -0.37$.

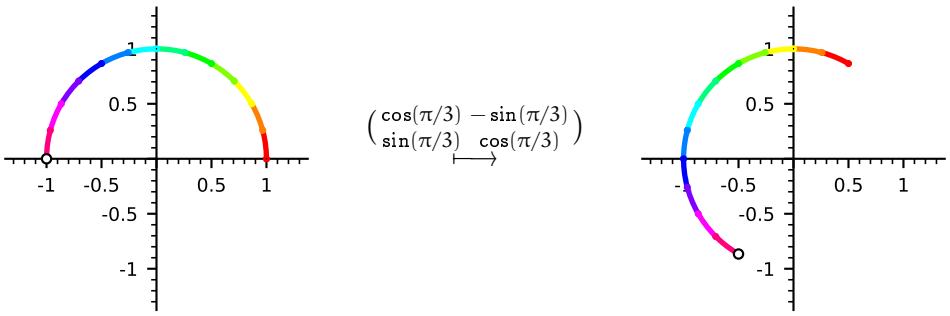



**Plane rotation** We know the $2\times 2$ matrix that rotates all vectors counterclockwise.

$$\begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$$

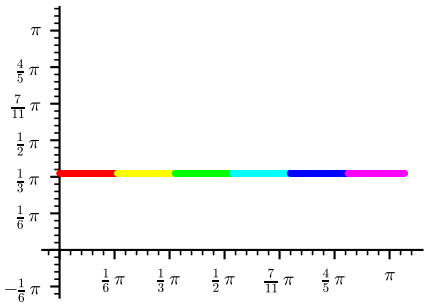
Here is its action on the upper half circle with $\theta = \pi/3$.

```
sage: load("plot_action.sage")
None
sage: q = plot_circle_action(1,0,0,1)
sage: q.set_axes_range(-2, 2, -2, 2)
None
sage: q.save("graphics/eigen200a.pdf")
None
sage: p = plot_circle_action(cos(pi/3),-sin(pi/3),sin(pi/3),cos(pi/3))
sage: p.set_axes_range(-2, 2, -2, 2)
None
sage: p.save("graphics/eigen200b.pdf")
None
```




This map turns every input vector by $\pi/3$ radians so its input/output angle graph has no interesting points.

```
sage: load("plot_action.sage")
None
sage: p = plot_color_angles(cos(pi/3),-sin(pi/3),sin(pi/3),cos(pi/3))
sage: p.set_axes_range(0,pi,-0.25*pi,pi)
None
sage: p.save("graphics/eigen201.pdf",  tick_formatter=[pi,pi])
None
```




This transformation has no real number eigenvalues.

## Matrix polynomials

*Sage* can find characteristic and minimal polynomials of a matrix.

```
sage: M =  matrix(RDF, [[1, 2], [3, 4]])
sage: poly = M.charpoly()
sage: poly.factor()
(x - 5.372281323269014) * (x + 0.3722813232690143)
sage: poly.roots()
[(-0.3722813232690143, 1), (5.372281323269014, 1)]
```

Of course, the eigenvalues are the roots.

Recall that the characteristic polynomial and minimal polynomial can 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^3 - 8.0*x^2 + 20.0*x - 16.0
sage: M.charpoly().factor()
(x - 4.0) * (x^2 - 3.9999999999999987*x + 4.000000000000001)
sage: M.minpoly().factor()
(x - 4.0) * (x^2 - 3.9999999999999987*x + 4.000000000000001)
```

*Sage* has trouble here telling whether 2 is a repeated root, because of floating point numerical issues. Doing an exact calculation using rational numbers resolves the question.

```
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
```

The rotation matrix has no real number eigenvalues but it does have complex eigenvalues.

```
sage: R = matrix(RDF, [[cos(pi/3), -sin(pi/3)], [sin(pi/3), cos(pi/3)]])
sage: R.minpoly()
x^2 - x + 1.0
sage: R.eigenvalues()
[0.5 + 0.8660254037844388*I, 0.5 - 0.8660254037844388*I]
sage: R.eigenvectors_right()
[(0.5 + 0.8660254037844388*I, [(-0.7071067811865475*I, -0.7071067811865476)], 1), (0.5 -
    0.8660254037844388*I, [(0.7071067811865475*I, -0.7071067811865476)], 1)]
```

## 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. (Even if you haven't covered this matrix form, the command is simple.)

```
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 horizontal blocks. Likewise, there are vertical lines.

*Sage* will also find the transformation matrix, the nonsingular P that can convert between the given matrix M and the matrix D of the desired form, with $M = PDP^{-1}$.

```
sage: M =  matrix(QQ, [[2, -2, 2], [0, 1, 1], [-4, 8, 3]])
sage: JF, P = M.jordan_form(transformation=True)
sage: JF
[3|0|0]
[-+-+-]
[0|2|0]
[-+-+-]
[0|0|1]
sage: P
[  1   1   1]
[1/2 4/9 1/2]
[  1 4/9   0]
sage: P^(-1)
[ -4   8   1]
[  9 -18   0]
[ -4  10  -1]
sage: P*JF*P^(-1)
[ 2 -2  2]
[ 0  1  1]
[-4  8  3]
```

## Source of plot_color_angles

This routine gathers graphic instances and returns the plot of that list. The default is to plot a thousand points.

```
    """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(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. The TICKS constant gives the places where the axes are labeled.

```
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 orientation. That suits the purpose here, which is to use the graph to roughly locate places where the action of the matrix does not turn the vector.

```
    """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 = M*v
        try:
            dot = v[0]*w[0] + v[1]*w[1]  # dot product
            det = v[0]*w[1] - v[1]*w[0]  # determinant
            angle = atan2(det, dot)      # atan2(y, x) or atan2(sin, cos)
        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

```
EPSILON = 0.25
```

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 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 = M*v
        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. https://hefferon.net/linearalgebra, 2021.

David Joyner and William Stein. Open Source Mathematical Software. *Notices of the AMS*, page 1279, November 2007.

Jupyter Team. Project Jupyter, 2021. URL https://jupyter.org/. [Online; accessed 2021-Sep-20].

Python Team. Floating point arithmetic: issues and limitations, 2021a. URL https://docs.python.org/3/tutorial/floatingpoint.html. [Online; accessed 2021-Sep-20].

Python Team. The Python Tutorial, 2021b. URL https://docs.python.org/3/tutorial/. [Online; accessed 2021-Sep-20].

Sage Development Team. Sage Tutorial 9.4. https://doc.sagemath.org/html/en/tutorial/, 2021a. [Online; accessed 2021-Sep-20].

Sage Development Team. Sage Reference Manual 9.4. https://doc.sagemath.org/html/en/reference/, 2021b. [Online; accessed 2021-Sep-20].

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

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

Wikipedia. The Great Wave off Kanagawa, 2019. URL https://en.wikipedia.org/wiki/The_Great_Wave_off_Kanagawa. [Online; accessed 19-Nov-2019].