Chapter 1

Welcome to Sage!

Mathematics is something that we do, not something that we watch. Therefore, the best way to learn any branch of mathematics is to actually *get started* and give things a try. The same is true for learning Sage! Just connect to the Sage "Cell Server," the simplest of several interfaces to Sage, and dive right on in! Experiment with the examples on the next few pages, trying them out as you read.

`https://sagecell.sagemath.org/`

1.1. Using Sage as a Calculator

First off, you can always use Sage as a simple calculator. For example, if you type `2+3` and click "Evaluate," then you will learn the answer is 5.

1.1.1. Arithmetic Operations in Sage

The expressions can become as complicated as you like. To calculate the amount on a simple interest loan at 6% per year for 90 days and principal \$900, we all know the formula to be $A = P(1 + rt)$ so we would just type in

```
900*(1+0.06*(90/365))
```

and click "Evaluate." You will learn that the answer is 913.315068493151, or \$913.32 after rounding to the nearest penny[1].

Notice that the symbol for addition is the plus sign (`+`) and the symbol for subtraction is the hyphen (`-`). The symbol for multiplication is the asterisk (`*`) and the symbol for division is the forward slash (`/`), not the backslash (`\`). On keyboards in the USA, the forward slash is found with the question mark (`?`), while the backslash is found with the vertical line or "pipe" (`|`). The parentheses, `(` and `)`, are used for grouping in Sage exactly as they are used in math, but we'll see that they have many other uses as well. These are the same symbols (called "operators") used by

[1]Some readers from outside the USA might not be familiar with the word "penny." It refers to one hundredth of a dollar, \$0.01. Many other countries use the same word, such as Canada.

MS-Excel and many computer languages, including C, C++, Perl, Python, MATLAB, FORTRAN, and Java. There is another operator for division, denoted `//`, that is not needed very often in mathematics. We will see a detailed description of it in Subsection 4.6.6, starting on page 219.

For compound interest, we'll need to take exponents. The symbol for exponents is the caret (`^`), sometimes called a "hat" or a "circumflex." On keyboards in the USA, the caret is found with the number `6`.

Let's consider 12% compounded annually on a signature loan for 3 years, and a principal of $800. Hopefully, we all know the formula to be

$$A = P(1 + i)^n$$

so we would just type in

```
800*(1+0.12)^3
```

and click "Evaluate." By the way, instead of clicking "Evaluate," you can just press shift-enter on the keyboard, and it will do the same thing. Either way, you will learn that the answer is 1,123.94240000000, or $1,123.94 after rounding to the nearest penny. The extra precision might be irksome to students of business and finance. We will learn how to avoid excessive precision on page 9 in Subsection 1.2.2.

For those who don't like the caret, you can use two asterisks in a row, to represent exponentiation.

```
800*(1+0.12)**3
```

which is actually a throwback to the programming language FORTRAN, which was most popular[2] during the 1970s and 1980s.

What if you make a mistake? Let's say that I really meant to say $900 and not $800. Click on the mistake, and correct it using the keyboard by changing the `800` to `900`. Now click "Evaluate" again, and all is well.

1.1.2. Vital: Representing Large Numbers in Sage

It is very important not to enter a comma inside large numbers when using Sage. You have to type `11000` and not `11,000`.

- Most of my readers are aware that in the vast majority of English-speaking nations, we write `12,345,678.90` in documents, and the same is done in several other countries.
- In the vast majority of French-speaking nations, that number would be written `12.345.678,90` instead, interchanging the positions of the period (`.`) and the comma (`,`) symbols.

[2] An irrelevant historical aside: Because rewriting software is a painful and time-consuming process, many important pieces of scientific software remain written in FORTRAN even in the year that this footnote was written (2020), and likewise the same is true of business software in COBOL. Each new programming language incorporates features of the previous generation of languages, to make it easier to learn, and accordingly one sees some truly ancient seeds of decades-old languages once in a while.

- In the vast majority of Spanish-speaking nations, a space (smaller than the space between consecutive words) is used as the "thousands separator." Therefore, that number would be written as `12 345 678.90` there.
- In Germany, that number could be written following either the French style or the Spanish style.
- In Russia, a space is also used as the thousands separator, but the comma (`,`) is used as a decimal point. Therefore, this number would be written as `12 345 678,90` instead.
- In India, that number would be written `1,23,45,678.90` because they group digits by twos (hundreds) instead of by threes (thousands), after the first grouping of three digits.
- In China, historically the grouping was by four digits (tens of thousands). This is still how numbers are spoken, but in writing the grouping by threes (thousands) is commonly used, especially in computing.
- In other countries, that number would be written `12345678,90` which has the thousands separators removed.
- In most programming languages today, that number is written `12345678.90` which also has the thousands separators removed.

Until Sage Version 9.0 (released in January 2020), Sage would also force you to type `12345678.90`, as is the case with the vast majority of computer languages. Now we have a new syntax available for those who want to group digits inside of large numbers, for readability. You can place underscores (`_`) anywhere inside a number, grouping the digits in such a way as you might feel most comfortable. The placement of the underscores is arbitrary. For example, each of the following represents the same number:

```
12345678.90
12_345_678.90
1234_5678.90
1_23_45_678.90
```

It is worth mentioning that you cannot have two or more consecutive underscores, you cannot start a number using an underscore, and you cannot end a number using an underscore.

1.1.3. Grouping Symbols

When there are multiple sets of parentheses in a formula, sometimes mathematicians use brackets as a type of "super parentheses." As it turns out, Sage needs the brackets, (`[`) and (`]`), for other things, like lists, so you have to always use parentheses for grouping inside of formulas.

For example, let's say you need to evaluate

$$550\frac{\left[1-(1+0.05)^{-30}\right]}{0.05}.$$

In Sage, you should not type

```
550 [ 1 - (1+0.05)^(-30) ]/0.05
```

but rather

```
550*( 1 - (1+0.05)^(-30) )/0.05
```

where the brackets have become parentheses. Note, we need the asterisk (`*`), to show the multiplication of the 550.

Some very old math books use braces, { and }, as a sort of auxiliary grouping symbol along with parentheses and brackets. These too, if they are for grouping in a formula, must become parentheses. As it turns out, Sage (as well as modern mathematical books), use the braces { and } to denote either sets or something called "Python Dictionaries," both of which won't be covered in this book. Sage has other uses for braces as well.

By the way, the above formula was not artificial. It is the value of a loan at 5% compounded annually for 30 years, with an annual payment of $550. The formula is called "the present value of an annuity."

1.1.4. The Print Command in Sage

You may have noticed that all of our computations in Sage so far have been one-line programs. As we tackle more and more sophisticated problems, we will have programs that are several lines long. On the other hand, even though I have been using Sage for many years, I remain surprised at the incredible breadth and depth of problems that can be solved in programs of only ten or fewer lines. I'm sure that you will agree before the end of Chapter 1. Let's now try a multi-line program.

It is common to use the `print` command in Sage to display intermediate steps. While Sage will always print the output of the last line, printing any other output requires a `print` command. Let's say that a banker will deposit $76,543.21 for 4 months in an account that has 3% interest, compounded monthly. He wishes to know how much he will have at the end of each month.

```
print(76_543.21*(1 + 0.03/12)^1)
print(76_543.21*(1 + 0.03/12)^2)
print(76_543.21*(1 + 0.03/12)^3)
print(76_543.21*(1 + 0.03/12)^4)
```

We get the following output, which is correct:

```
76734.5680250000
76926.4044450625
77118.7204561751
77311.5172573156
```

Those of you who have programmed before know that we can make this program far more flexible and more readable with something called a `for` loop. We'll learn about that in Subsection 5.1.1, starting on page 319. Next, you might want to hide the irrelevant decimal places past the pennies. The

second-least significant digit in those numbers above (namely the 0, 2, 5, and 5) represent "nano-dollars," or one billionth of a dollar. We'll learn about hiding excessive precision on page 9 in Subsection 1.2.2.

1.1.5. A Reminder about the Order of Operations

By this point in your explorations of mathematics, you're already an expert in the order of operations. When applied to computer programming, the order of operations is called "operator precedence," a commonly used technical term among computer scientists. However, there are two cases that I've found have confused some students in the past.

First, consider the code

```
print( -8^2 )
print( (-8)^2 )
```

which produces the output

```
-64
64
```

Many students are surprised that the output is not the same. We should remember that -8 is actually an abbreviation of convenience, for $0 - 8$. Therefore,

$$-8^2 = 0 - 8^2 = 0 - (8)(8) = 0 - 64 = -64$$

which contrasts with

$$(-8)^2 = (0 - 8)^2 = (-8)(-8) = 64.$$

Keep in mind that the distinction we've made here is true in all of mathematics, not just in Sage. Next, if we want to compute in Sage

$$\frac{11 + 4}{9 + 16} = \frac{15}{25} = \frac{3}{5},$$

then we must remember that there's an implied (meaning, invisible) pair of parentheses around the numerator and the denominator.

To explain this, we should look at the code

```
print( 11 + 4 / 9 + 16 )
print( (11 + 4) / (9 + 16) )
```

which produces the output

```
247/9
3/5
```

As you can see, it is the second line, where the implied or invisible parentheses of written mathematics were explicitly added, that worked. In contrast, the first line of that code means this:

$$11 + \frac{4}{9} + 16 = \frac{11(9)}{9} + \frac{4}{9} + \frac{16(9)}{9} = \frac{99}{9} + \frac{4}{9} + \frac{144}{9} = \frac{99 + 4 + 144}{9} = \frac{247}{9}.$$

In closing, note that this not a special feature of Sage or Python. These same distinctions apply when using C, Java, BASIC, Pascal, or even Microsoft

Excel, as well as doing mathematics on the blackboard or whiteboard. When in doubt, you can always remove ambiguity by adding parentheses where needed.

1.1.6. Labelling Output with Character Strings

It is often extremely useful to label the output of a computer program. We can do that as follows:

```
print( 'First month:', 76_543.21*(1 + 0.03/12)^1 )
print( 'Second month:', 76_543.21*(1 + 0.03/12)^2 )
print( 'Third month:', 76_543.21*(1 + 0.03/12)^3 )
print( 'Fourth month:', 76_543.21*(1 + 0.03/12)^4 )
```

We get the following output, which is nice and readable:

```
First month: 76734.5680250000
Second month: 76926.4044450625
Third month: 77118.7204561751
Fourth month: 77311.5172573156
```

Whatever we place between the two apostrophes (') or "single quotes" is formally called a character string, but everyone just calls it a string. (The term string is used in essentially all programming languages.) A string is a sequence of characters: letters, numbers, spaces, punctuation marks, or other symbols. Above, we used strings to label our output, but strings have other uses too.

The "quotation marks" used in English text are usually double quotes ("), and double quotes are used in several famous programming languages, such as C, C++, and Java. In Sage and Python, you are free to use either apostrophes (single quotes) or quotation marks (double quotes), as you might prefer. For example,

```
print( "First month:", 76_543.21*(1 + 0.03/12)^1 )
```

is perfectly valid code. Nonetheless, it seems that most Sage programmers use single quotes (').

1.1.7. A Challenge: Modelling Savings

I'm going to challenge you with a task in each section—often at the end, but sometimes in the middle of the section. These challenges should not be skipped, because they will help cement your recently acquired knowledge. I'd like you to mimic the code that we wrote for the banker on page 6 in Subsection 1.1.6, but change it to solve the following rather similar problem:

> A college student, at the end of her first year, has received an inheritance of $23,456.78 from an uncle whom she had never met. She has no use for this money at the moment (because she has a full scholarship), so she decides to save it, using a money market account that pays 4% compounded quarterly.

She will graduate exactly three years later. How much money does she have at the end of each year?

By the way, in almost all cases, I will not provide a solution to the challenge problem. That's to enable instructors to use these questions as homework. In almost all cases, you will be able to tell very easily whether you've gotten the challenge correct. This problem is an exception, because the answer is "an ugly decimal" (an arbitrary real number). The final answer at graduation is 26,431.686830298984704, which we'd normally round to $26,431.69. Of course, you still need to find the intermediate answers.

1.2. Using Sage with Common Functions

Now I'll discuss how Sage works with square roots, logarithms, exponentials, and so forth. We'll also discuss some important tools for working with decimals, such as scientific notation, significant figures, and constants. I also have two challenges for you. One is financial, and one is mathematical.

1.2.1. Square Roots

The standard "high school" functions are all built into Sage. For example, if you type

```
sqrt(144)
```

and then click "Evaluate," then you'll learn that the answer is `12`. From now on, I'm not going to say "click Evaluate" because repeating that might get tiresome—I'll assume that you know you have to do that. Since Sage likes exact answers, try

```
sqrt(8)
```

and get `2*sqrt(2)` as your answer. If you really need a decimal, try instead

```
N( sqrt(8) )
```

and obtain

```
2.82842712475
```

which is a decimal approximation.

The `N()` function, which can also be written `n()`, will convert what is inside the parentheses to a real number, if possible. Usually that is a decimal expansion. Thus unless what is inside the parentheses is an integer[3], then it will be, necessarily, an approximation. Sage will assume that all decimals are mere approximations, so for example

```
sqrt(3.4)
```

[3]To be mathematically correct, I should say "an integer or a fraction with denominator writable as a product of a power of 5 and a power of 2." For example, 25ths, 16ths, and 80ths can be written exactly with decimals, whereas 3rds, 15ths, and 14ths cannot. Observe that $25 = 5^2$ and $16 = 2^4$ as well as $80 = 2^4 \times 5$. Those denominators have only 2s and 5s in their prime factorization. Meanwhile, $3 = 3$ and $15 = 5 \times 3$ while $14 = 7 \times 2$. As you can see, those denominators have primes other than 2 and 5 in their prime factorization. If you find this intriguing, then you should read "Working with the Integers and Number Theory," Section 4.6, starting on page 213.

will evaluate to `1.84390889145858`, without need of using `N()`. Similarly, we saw that `sqrt(144)` resulted in `12`, but instead

```
sqrt(144.0)
```

results in

```
12.0000000000000
```

1.2.2. Significant Digits and Numerical Approximation

In Sage, if you type

```
numerical_approx(sqrt(2), digits=200)
```

then you will get a decimal approximation of $\sqrt{2}$ that has 200 significant digits (sometimes called "significant figures"). You can also type

```
N(sqrt(2), digits=200)
```

instead, because `N()` and `n()` are just abbreviations for the command `numerical_approx()`.

A Crucial Distinction:

It is important to realize that "precise to four significant digits" and "precise to four decimal places" are not at all the same thing. For example,

```
print( N(12.3456, digits=4) )
print( N(0.123456, digits=4) )
print( N(0.0000123456, digits=4) )
```

produces the output

```
12.35
0.1235
0.00001235
```

each of which is precise to four significant digits. Yet, the first is showing two decimal places, the second is showing four decimal places, and the last is showing eight decimal places.

Of course, there's never any reason to be stressed out about this distinction, because if you don't get the output that you desired, you can just change the `digits=` number to something else and click "Evaluate" again. I personally do that sort of "fine tuning of the output" very often. With Sage, there simply is no need to "get it right on the first try."

The Special Constants π and e:

Frequently in math we need the special constants π and the base of the natural logarithm:

$$e \approx 2.718281828\ldots.$$

It turns out that π is built into Sage as "`pi`" (both letters lowercase) and e is built in as "`e`" (again, lowercase).

You can do things like

```
N(pi, digits=200)
```

to get a decimal expansion of pi, with 200 significant digits.

The golden ratio

$$\varphi = \frac{1+\sqrt{5}}{2} \approx 1.618033988\ldots$$

that comes up in geometry, architecture, and the Fibonacci numbers, and which was a rather trendy[4] topic of conversation at the turn of the century, is also built into Sage. You can get 200 digits of it with

```
N(golden_ratio, digits=200)
```

Controlling Excessive Precision

Back on page 5 in Subsection 1.1.4, we had expressed frustration that certain dollar amounts were given with ten decimal places. Even the second-to-last digits represented "nano-dollars" or billionths of a dollar, in that example. Indeed, throughout the USA it is a standard practice to report all[5] financial amounts below one million dollars to the penny, if possible, but essentially never more precise than that.

Now, we can fix the excessive precision by using the `N()` function. Here is the code to do that:

```
print('First month:', N(76_543.21*(1 + 0.03/12)^1, digits=7) )
print('Second month:', N(76_543.21*(1 + 0.03/12)^2, digits=7) )
print('Third month:', N(76_543.21*(1 + 0.03/12)^3, digits=7) )
print('Fourth month:', N(76_543.21*(1 + 0.03/12)^4, digits=7) )
```

However, I have to sometimes fiddle with the number of digits. For example, if the dollar amounts went into the hundreds of thousands, I'd have to use `digits=8` to get Sage to show the answer to the nearest penny, which is the standard practice in finance. For example,

```
N( 123_456.987654321, digits = 7 )
```

results in

```
123457.0
```

and I need to set `digits=8` to get that second decimal place.

---

[4] Primarily due to prominence in the fictional but fascinating mystery novel *The Da Vinci Code*, by Dan Brown, published in 2003 by Doubleday [**13**].

[5] The one exception (in the USA) where additional precision is used happens to be gasoline prices, which are reported to the tenth of a penny, or equivalently a thousandth of a dollar. This is weird because there is no coin in the USA smaller than the penny, since the abolition of the half-penny coin in 1857.

An Odd Feature Regarding π:

Any computation involving `pi` will automatically revert to symbolic mode. For example, the code

```
sqrt(144.0*pi)
```

results in the output

```
12.0000000000000*sqrt(pi)
```

which is surprising. To get a decimal, we should instead type

```
N( sqrt(144.0*pi) )
```

and that results in the output

```
21.2694462108662
```

There are good reasons for this nuance, related to Fourier transforms—an advanced topic in mathematics which cannot be explained well in only a few sentences. Suffice it to say that Sage is good at Fourier transforms, which are essential in signal processing, but which have other applications too, such as in computational thermodynamics. Unfortunately, we cannot go into Fourier transforms in this book.

1.2.3. Scientific Notation

Suppose that I wanted to use Avogadro's constant, $6.02214076 \times 10^{23}$ or Newton's gravitational constant $G = 6.674 \times 10^{-11}$, in some Sage computation dealing with chemistry or physics. I really don't want to type

```
0.000_000_000_066_74
```

because that would be accident-prone, and it is hard to read. Also, it doesn't look like proper scientific writing. Indeed, that particular style of typing that number would have been even harder to write prior to the introduction of the underscore (`_`) notation into Sage, which occurred in January 2020 and that we saw in Subsection 1.1.2, starting on page 2. Without the underscore, I would have had to type

```
0.00000000006674
```

which is really accident-prone. Think about how easy it is to accidentally omit a zero, or inadvertently include a twelfth zero.

Luckily, scientific notation in Sage works exactly like the way you would write it with a pencil. Here is an example:

```
print("Avogadro's constant:", 6.02214076*10^23)
print("Newton's gravitational constant:", 6.674*10^-11)
```

produces the output

```
Avogadro's constant: 6.02214076000000e23
Newton's gravitational constant: 6.67400000000000e-11
```

The careful reader will note that I used the double quotes, or quotation marks (`"`), instead of the single quote (`'`), even though I always prefer the single quote. The reason is that I cannot have an apostrophe (') in "Avogadro's" or "Newton's" when using the single quote. Alternatively, I can also type

```
print('Avogadro\'s constant:', 6.02214076*10^23)
```

where the backslash (\) has been used to tell Sage that the apostrophe in Avogadro's name is not the end of the string, but merely an apostrophe. I do not recommend using the backslash in this way, because using double quotes is much more readable.

I should share with you that you can type `6.02214076e23` in place of `6.02214076*10^23` in Sage but that I do not recommend doing so. That abbreviation only saves you three keystrokes, and it risks a serious misunderstanding. It is easy to imagine that lowercase `e` is the base of the natural logarithm, $e \approx 2.718281828$, but when used in that way, in the middle of a number, `e` does not mean that at all.

In other words,

$$\texttt{6.02214076e23} = 6.02214076 \times 10^{23} \neq 6.02214076e^{23}$$

and I have seen more than a few students be confused on this point.

1.2.4. Is Sage Case-Sensitive?

You've probably had a situation in life where you entered your password correctly but discovered that it was rejected because you had the wrong capitalization. For example, perhaps you've left the "CAPS-LOCK" key on. In any event, Sage does think of `Pi` as different from `pi`, `E` as different from `e`, and `Print` as different from `print`.

The only four exceptions known to me are `i` and `n()`, as well as `True` and `False`. We will not make extensive use of `True` and `False` until Chapter 5, but it is harmless to mention that you can also enter them as `true` and `false`, and Sage recognizes[6] these as the same. You can write `n(sqrt(2))` or `N(sqrt(2))` and Sage treats those as identical.

Lastly, you may have heard that $\sqrt{-1}$ is often referred to as "the imaginary number" and is denoted i. You can represent $\sqrt{-1}$ as either `i` or `I` and either way Sage will know what you meant, namely $\sqrt{-1}$. If you've never heard of complex/imaginary numbers or i, then you can safely ignore this fact. We'll talk briefly about complex/imaginary numbers in Subsection 1.2.10, starting on page 14.

So far as I am aware, these easements only apply to $i = \sqrt{-1}$ and `n()`, as well as `True` and `False`—in all other cases, capitalization matters.

[6]For completeness, I should add that Python itself requires `True` and `False` to be capitalized, but that does not affect Sage.

1.2.5. Exponential Functions

Just as `2^3` gives you 2^3 and likewise `3^3` gives you 3^3, if you want to say e^3, then just type

```
e^3
```

and that's fine. Or for a decimal approximation you can do

```
N(e^3)
```

Also, it is worth mentioning that sometimes books will write

$$\exp(5 \cdot 11 + 8)$$

instead of

$$e^{5 \cdot 11 + 8}$$

and Sage thinks that's just fine. For example,

```
exp(5*11+8) - e^(5*11+8)
```

evaluates to 0. Don't forget the asterisk between the 5 and the 11.

1.2.6. Logarithms

Of course, Sage knows about logarithms—but there's a problem. There are several types of logarithms, including the common logarithm, the natural logarithm, the binary logarithm, and the logarithm to any other base you feel like using.

In high schools in the USA, "log" refers to the common logarithm, and "ln" to the natural logarithm. However, in most courses above calculus, "log" refers to the natural logarithm. Since Sage was mainly meant for university-and-higher level work, then it is only natural that developers chose to use the natural logarithm for `log`.

- To find the natural logarithm of 100, often denoted $\ln 100$ or $\log_e 100$, just type
  ```
  N( log(100) )
  ```
- For the common logarithm of 100, often denoted $\log_{10} 100$, type
  ```
  N( log(100, 10) )
  ```
- For the binary logarithm of 100, often denoted $\log_2 100$, type
  ```
  N( log(100, 2) )
  ```
- Of course, this notation generalizes. For example, to find the logarithm of 100 taken in base 42, often denoted $\log_{42} 100$, type
  ```
  N( log(100,42) )
  ```

Note that Sage is excellent at getting exact answers. Try

```
log( sqrt(100^3), 10)
```

and you will obtain 3, the exact answer to the problem

$$\log_{10} \sqrt{100^3}\,.$$

1.2.7. A Financial Example with Logarithms

Suppose you deposit \$5,000 in an account that earns 4.5% compounded monthly. Perhaps you are curious about when your total will reach \$7,000. Using the formula $A = P(1+i)^n$, where P is the principal, A is the amount at the end, i is the interest rate per month, and n is the number of compounding periods (number of months), we have

$$\begin{aligned} A &= P(1+i)^n, \\ 7{,}000 &= 5{,}000(1+0.045/12)^n, \\ 7{,}000/5{,}000 &= (1+0.045/12)^n, \\ 1.4 &= (1+0.045/12)^n, \\ \log 1.4 &= \log(1+0.045/12)^n, \\ \log 1.4 &= n\log(1+0.045/12), \\ \frac{\log 1.4}{\log(1+0.045/12)} &= n. \end{aligned}$$

Therefore, we type into Sage

```
log(1.4)/log(1+0.045/12)
```

and get the response `89.8940609330801`, so that we know 90 months will be required. Therefore we write the answer 90 months, or even better, 7 years and 6 months.

This is an example of a computer-assisted solution to a problem. In addition to that approach, Sage is also willing to solve the problem from start to finish, with no human intervention. We'll see that in Subsection 1.9.1, starting on page 65.

1.2.8. Three Mistakes That Are 80+% of Beginner Errors in Sage

Momentarily, I am going to present you with some challenges for you to solve in Sage, so it's a good time for me to point out some common pitfalls, so that you can avoid them. There are three mistakes that beginners frequently make when using Sage. For example, if you want to say $11{,}000x + 1{,}200$, then you have to type

```
11000*x + 1200
```

The first error that I sometimes see is that a beginner might leave out the asterisk between `11000` and `x`. In Sage, you must include that asterisk, as that is the symbol of multiplication. Actually, I still make this mistake myself, especially when tired.

The second error is that I'll often see a comma inside the `11000`, because my students are in the USA, and that's how we group digits here. However, it is not acceptable in Sage to write `11,000` for `11000`. Instead, we can write `11_000` or `11000` depending on one's preferences.

The third error is to have mismatched parentheses. Any Sage expression should have the same number of (’s as it has of)’s—no more and no fewer.

A more complete list is given as Appendix A, starting on page 433, “What to Do When Frustrated!”

1.2.9. A Pair of Challenges with Logarithms

We’ve had several finance problems in a row, so perhaps now is a good time to try something else. I’m going to give you two challenges now, and you can pick the one that you prefer.

- Using logarithms taken base 68, find the value of x that solves this equation:

$$314{,}432\left(68^{x/2-3}\right) = 21{,}381{,}376\left(68^{1+3x/4}\right).$$

- Following the pattern of Subsection 1.2.7, starting on page 13, solve the following financial problem. All details are the same, except that the deposit is \$6,000, the interest rate has dipped to 3.75%, and the customer wants to know how long it will be until their investment reaches \$8,500.
- In both challenges, you can easily check your work to see if you have the right answer, by plugging back into the original formula. Most challenges in this book will work that way.

1.2.10. Square Roots and Imaginary/Complex Numbers

Here are a couple of notes about square roots that would appeal to math majors, or anyone who wants to work with the complex numbers. Most students will want to skip this subsection and continue with either “Using Sage for Trigonometry” (Section 1.3, starting on page 16) or with “Using Sage to Graph 2-Dimensionally” (Section 1.4, starting on page 21).

Though it might sound rather pompous, you surely know that $(-2)^2 = 4$ as well as $2^2 = 4$. So technically, we should say that *both* -2 and 2 are square roots of 4. If you want *all* the square roots of a number, you can type

```
sqrt(4,all=True)
```

to obtain

```
[2, -2]
```

where the square brackets ([and]) indicate a list. Any sequence of numbers separated by commas and enclosed in square brackets is a list. We’ll see other examples of lists throughout the book.

If you know about complex numbers, you can use

```
sqrt(-4,all=True)
```

to obtain

```
[2*I, -2*I]
```

where the capital letter I represents $\sqrt{-1}$, the imaginary constant. As I mentioned before, you can use the lowercase `i` instead of the capital I, if you prefer.

For example, you can type

```
(2+i) * (2-i)
```

and learn that the answer is 5, which we can verify manually with the following calculation:

$$(2+i)(2-i) = 4 - 2i + 2i - i^2 = 4 - i^2 = 4 - (-1) = 4 + 1 = 5.$$

1.2.11. Higher-Order Roots

Just as a square root can be represented as the 1/2 power, a cube root is the 1/3 power, a fourth root is the 1/4 power, and a fifth root is the 1/5 power. You can type

```
64^(1/6)
```

to learn that the sixth root of 64 is 2, which we write mathematically by

$$\sqrt[6]{64} = 2.$$

For negative numbers, there is a subtle nuance. Just as 2 and -2 are the two square roots of 4 and both $2i$ and $-2i$ are the two square roots of -4, it turns out that all non-zero real numbers have three different cube roots. Furthermore, one cube root is real, and the other two cube roots are complex. (In contrast, zero is the unique cube root of zero.)

For very good reasons, which will be explained in Subsection 4.26.5, starting on page 309, Sage will return an imaginary root when you ask for the cube root of a negative number. That does present some problems, especially when trying to graph the cube-root function. (We'll talk about that in Subsection 1.4.4, starting on page 26.) To circumvent those difficulties, I successfully lobbied for a new command to be added to Sage, and it is called `real_nth_root`. This command has been available in Sage since Version 9.2. While `(-32)^(1/5)` will return a complex number, the new command `real_nth_root` can be used to return a real number.

The code

```
print( (-32)^(1/5) )
print( real_nth_root( -32, 5 ) )
print( N( (-32)^(1/5) ) )
print( N( real_nth_root( -32, 5 ) ) )
```

produces the output

```
2*(-1)^(1/5)
-2
1.61803398874989 + 1.17557050458495*I
-2.00000000000000
```

As you can see from the second-to-last line of the output, `(-32)^(1/5)` returned a complex number. However, `real_nth_root( -32, 5 )` returned a real number, -2, as you can see from the last line. If you're curious about the process of getting a command added into an open-source project like Sage, then you might enjoy reading Subsection 4.26.7, starting on page 312.

1.3. Using Sage for Trigonometry

Some students are unfamiliar with trigonometry, or are in a course that has nothing to do with trigonometry. If so, then they may confidently skip this section and jump to the next one, "Using Sage to Graph 2-Dimensionally" (Section 1.4, starting on page 21).

The commands for trigonometry in Sage are very intuitive but it is important to remember that Sage works in radians. So if you want to know the sine of $\pi/3$, you should type

```
sin(pi/3)
```

and you will get the answer `1/2*sqrt(3)`. This is an exact answer, rather than a mere decimal approximation. You will find that Sage is very oriented toward exact rather than approximate answers. Sometimes this is irritating because if you ask for the cosine of $\pi/12$, then you would type

```
cos(pi/12)
```

and obtain `1/4*sqrt(6) + 1/4*sqrt(2)`, which is especially unsatisfying if you want a decimal. Instead, if you type

```
N(cos(pi/12))
```

then you will obtain `0.965925826289`, a rather good decimal approximation.

You will discover that Sage is fairly savvy when it comes to knowing when functions will go wrong. In particular, just try evaluating tangent at one of its asymptotes. For example,

```
tan(pi/2)
```

will produce the helpful answer "Infinity." The "rare" or "reciprocal" trigonometric functions (namely cotangent, secant, and cosecant), which are important in calculus but annoying on handheld calculators, are built into Sage. They are identified as `cot`, `sec`, and `csc`.

The inverse trigonometric functions are also available. They are used just like the trigonometric functions. For example, if you type

```
arcsin(1/2)
```

you will obtain

```
1/6*pi
```

as expected. Likewise

```
arccos(1/2)
```

produces

```
1/3*pi
```

The usual trigonometric abbreviations are all known by and used by Sage. Here is a complete list:

Math Notation:	Long-form Command:	Short-form Command:
$\sin^{-1} x$	`arcsin(x)`	`asin(x)`
$\cos^{-1} x$	`arccos(x)`	`acos(x)`
$\tan^{-1} x$	`arctan(x)`	`atan(x)`
$\cot^{-1} x$	`arccot(x)`	`acot(x)`
$\sec^{-1} x$	`arcsec(x)`	`asec(x)`
$\csc^{-1} x$	`arccsc(x)`	`acsc(x)`

You can also use Sage to graph the trigonometric functions. We'll do that in Subsection 1.4.2, starting on page 24.

1.3.1. Converting between Degrees and Radians

We all remember from trigonometry class that to convert radians to degrees just multiply by $180/\pi$, and to convert from degrees to radians just multiply by $\pi/180$. Accordingly, here's what you type to convert $\pi/3$ to degrees:

```
(pi/3) * (180/pi)
```

produces `60` while

```
60 * (pi/180)
```

produces `1/3*pi`.

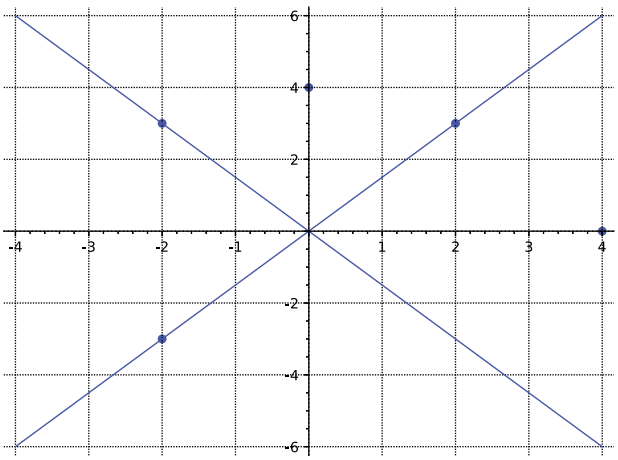
The way I like to remember this technique is that a protractor is 180 degrees, and the word "protractor" begins with "p," while π is the Greek letter "p." For example, 90 degrees is half a protractor, and 60 degrees is a third of a protractor. That's why I can remember that they are $\pi/2$ and $\pi/3$. Likewise, if I see $\pi/6$, then I know that this is one-sixth of a protractor or 30 degrees.

1.3.2. Another Way of Computing Arctangent

Consider the plot on page 18.

You can see that there are five points. Sometimes, we might want to know the angle between the x-axis and the line connecting a specific point to the origin. For example, this is done when converting from rectangular coordinates to polar coordinates.

Moreover, it comes up in other applications, such as computer graphics. Imagine an architect using computer-aided design (CAD) software to give a customer a tour of a new building, before construction has begun. You can



imagine a person standing at the origin, with objects at the locations where I have placed points. We need to know and compare those angles (between the x-axis and the line connecting the point to the origin) as well as the angle that the person is facing, in order to determine which objects to draw on the screen, and which are outside the field of view. (If you don't think that polar coordinates or computer graphics will be important applications for you, then you can skip to this section's challenge, in Subsection 1.3.3, starting on page 20.)

The standard way of doing this task is with an arctangent. Consider the point $(2, 3)$, as shown in the graph. Since tangent is "opposite leg over adjacent leg," we know that

$$\tan\theta = \frac{3}{2}$$

so we can find θ easily with

$$\theta = \arctan\frac{3}{2} = 0.982793723\ldots$$

in radians, which turns out to be $\approx 56.3099°$. That seems reasonable to my eye, looking at the graph.

Sometimes things go wrong, however. Consider the point $(-2, 3)$, as shown in the graph. The same calculation gives us

$$\theta = \arctan\frac{3}{-2} = \arctan -\frac{3}{2} = -0.982793723\ldots$$

in radians, which turns out to be $\approx -56.3099°$. Yet, we probably think of that angle as $\approx 123.690°$, in the sense of imagining a moving lever, starting at the origin and lying along the positive half of the x-axis, rotating counter-clockwise, passing the y-axis, and finally resting along that line. Of course, the lever could move in the opposite direction, rotating significantly less, and also align with that line but it would be pointing along the half of the line without $(-2, 3)$. For both polar coordinates and computer graphics, this is not what we had in mind.

Now consider the point $(-2, -3)$. For sure, it is undeniable that it is located on exactly the same line as the point $(2, 3)$, so we will get the same answer for $(-2, -3)$ as we obtained for $(2, 3)$. Specifically,

$$\theta = \arctan \frac{-3}{-2} = \arctan \frac{3}{2} = 0.982793723\ldots$$

in radians, which is the same as the first arctangent example, above. For the computer graphics application, it is really important to distinguish between $(2, 3)$ and $(-2, -3)$. Both points cannot be in the person's field of view at the same time, if the viewer is standing at the origin.

Even worse is the point $(0, 4)$, because we would compute

$$\theta = \arctan \frac{4}{0} = \text{division by zero!}$$

The solution to these difficulties goes back all the way to FORTRAN if not earlier. It is an alternative command for computing arctangents. Instead of `atan()`, there is a command `atan2()`. To compute the required angles (in degrees), use the code

```
print( N( atan2(0,4)*180/pi ) )
print( N( atan2(3,2)*180/pi ) )
print( N( atan2(4,0)*180/pi ) )
print( N( atan2(3,-2)*180/pi ) )
print( N( atan2(-3,-2)*180/pi ) )
```

which produces the output

```
0.000000000000000
56.3099324740202
90.0000000000000
123.690067525980
-123.690067525980
```

A few details are worth noting. First, the angle is defined by imagining a lever, anchored at the origin, initially aligned with the positive half of the x-axis and rotating until pointing towards the given coordinates. Counter-clockwise rotation, representing positive y-coordinates, is a positive angle; clockwise rotation, representing negative y-coordinates, is a negative angle. The careful reader will notice that the coordinates are reversed from normal notation. That's because `atan2(3,-2)` is intended as a replacement[7] for `atan(3/-2)`, even though we would normally write that point as $(-2, 3)$.

Furthermore, remember that Sage operates in radians. Therefore, I had to insert `*180/pi` into the code to get degrees, a unit of measure which I personally prefer. A huge advantage of `atan2()` is that there is no risk of

[7]Whether this notation is wise or unwise is not up for discussion. I have traced this notation back to FORTRAN-77, a standard for FORTRAN that was published in 1977, the year that I was born. There is a good chance that this notation existed in even earlier versions of FORTRAN as well.

division by zero in the case of the point $(0, 4)$, or any other point along the y-axis. The only problematic case is `atan2(0,0)`, where no angle would make any sense at all. In that case, Sage responds with `NaN`, an acronym that stands for "Not a Number."

Technically speaking, I should add that `NaN` is an "error code" and not an "error message." The distinction is that `NaN` does not halt the execution of a Sage program, whereas dividing by zero will halt a Sage program, unless particular steps are taken to prevent such halting. Those preventative steps, a topic called "exception catching" or "exception handling," involve the commands `try` and `except` and will be briefly described in Subsection 5.9.8, starting on page 409.

1.3.3. A Challenge: The Law of Cosines

This challenge deals with the Law of Cosines. If you've never heard of the Law of Cosines, then you're in for a real treat. It is as if the Pythagorean Theorem received a software upgrade and now can apply to *all* triangles, instead of only to *right* triangles. You can pick any of the three interior angles of the triangle for θ. The side opposite θ has length c, while the other sides have length a and b. Here is the formula:

$$c^2 = a^2 + b^2 - 2ab\cos\theta.$$

Before I get to the actual problem, to verify that I'm not making this up, plug $\theta = 90^\circ = \pi/2$ into that equation. Since $\cos\pi/2 = 0$, the Law of Cosines simply becomes the Pythagorean Theorem. However, the power of the Law of Cosines is that we can handle θ's that are not 90° or $\pi/2$ radians. Let's now use the Law of Cosines to solve a realistic problem.

Suppose Percy has an internship at his father's yacht club. There are some large trees at the sides of the lake where the yacht club is situated, and for a Halloween[8] event, the yacht's board wants to string some small electric lights to light up the lake. Two giant "weeping willow" trees are conveniently available. In order to know what length of string-lights to purchase, the distance between the trees must be known. After all, if the string of lights is too long, then it will sag into the water and short circuit—or if the string of lights is too short, then it won't even reach the other tree.

Of course, Percy clearly cannot take a tap measure and walk across the lake to measure the distance. Instead, he paces out a distance from one tree to a particular lamppost in the parking lot, and he counts 72 steps.

[8]For my readers outside the USA, Halloween is a holiday celebrated in the USA and some other countries, on the evening of the 31st of October, where small children dress in strange costumes, watch scary movies about ghosts and haunted houses, and acquire large amounts of candy.

Then, he paces out a distance from that same lamppost to the other tree, and he counts 83 steps. He carefully measures his step and it is 58 cm. Standing at the lamppost, he sees that the angle swept by looking from one tree to another is something like 110° to 115°. How far apart are the trees? Since you have an interval for the angle (which is perfectly realistic, since measuring angles is somewhat difficult), your answer for the distance between the trees should also be an interval, in meters.

Since the answer to this challenge is a "big ugly decimal," I feel as though I have to provide the answer. In this case, the answer is between 73.7326 m and 75.8983 m, so Percy would say "between 73.7 and 75.9 meters," or perhaps "between 73 meters and 76 meters."

1.4. Using Sage to Graph 2-Dimensionally

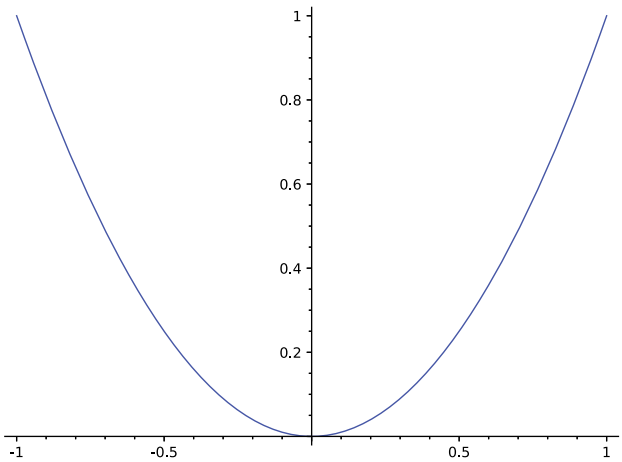
My favorite shape is the parabola, so let's start there. Type

```
plot(x^2)
```

and you get a lovely plot of a parabola going in the range

$$-1 < x < 1$$

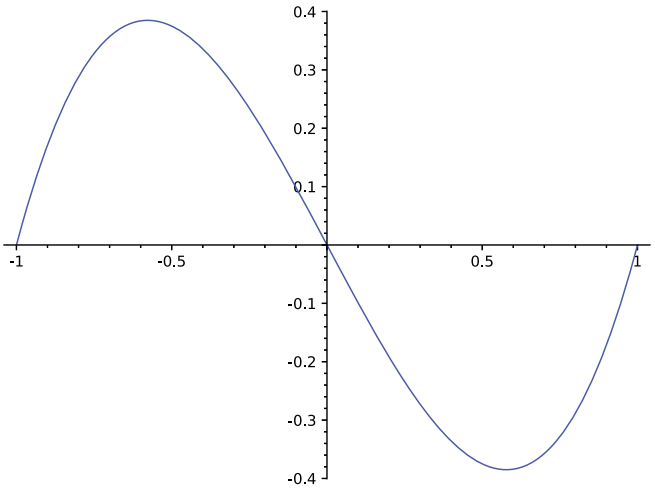
which is the default range for the `plot` command. It will bound the y-values to whatever is needed to show those x-values. Here's a screenshot of what you should see:



Likewise you can do

```
plot(x^3-x)
```

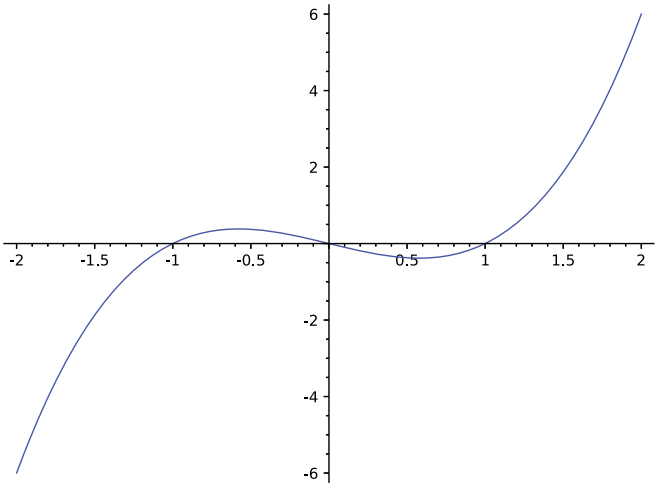
which is nice and visually appealing, as you can see:



What if you wanted a different x range? For example, to graph in $-2 < x < 2$ you would type

```
plot(x^3-x, -2, 2)
```

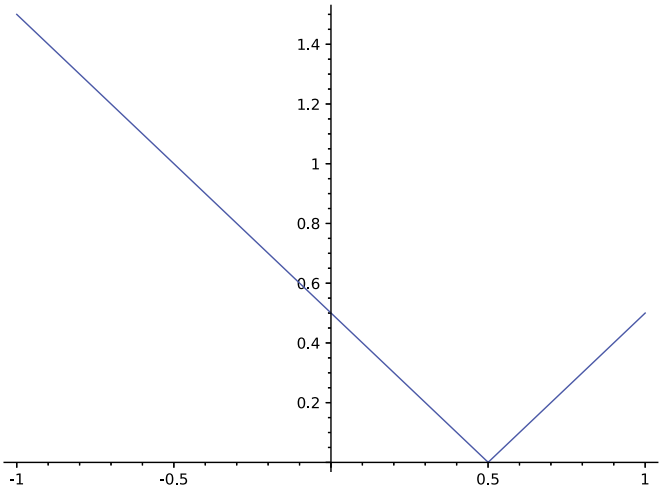
and you get the desired graph, namely:



For $|x - 1/2|$ you can do

```
plot(abs(x-1/2))
```

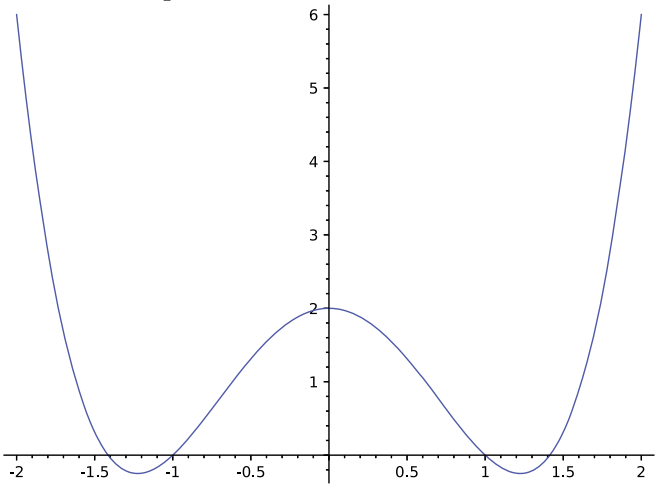
which produces the following plot:



A very cool graph is

```
plot(x^4 - 3*x^2+2,-2,2)
```

but notice the asterisk between 3 and x in " `3*x` ". You will get an error if you leave that out! The plot is



1.4.1. Saving a Plot as a File

You can save any of these plots to a file after having Sage generate them. Just right-click on the displayed graph in your web browser, and save the file to your computer.

You can then import the image into any report or paper that you might be writing. You can also attach it to an email, or even use it as your background image. If you'd like a higher quality ("vector graphics") format, then you can create an `eps` file. We'll discuss that in Subsection 1.13.4, starting on page 95.

1.4.2. Graphing Trigonometric Functions

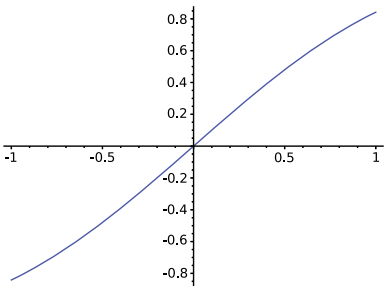
Let's say that you want to plot $y = \sin(t)$ for some reason. You have to say

```
plot(sin(x))
```

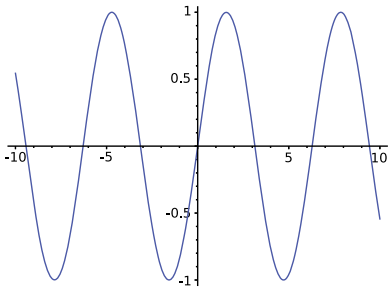
or better yet

```
plot(sin(x),-10,10)
```

because plot is not expecting t: it expects x. The images that you get are



plot(sin(x))



plot(sin(x),-10,10)

In fact if you were to type

```
plot(sin(t))
```

you would see

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

because in this case, Sage does not know what "`t`" means.

As you will learn a bit later, we will very often declare our own variables (typically lots of them), when using Sage, no matter what the application. The declaration of variables will be explained in detail in Subsection 1.8.2, starting on page 55.

1.4.3. Controlling the Viewing Window of a Plot

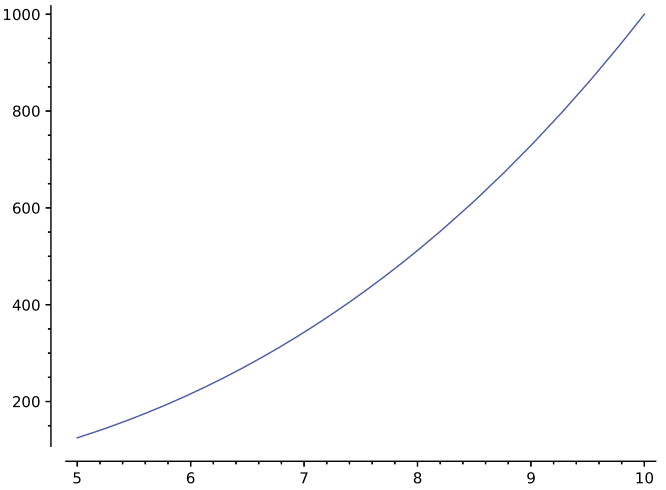
Much of the time, the default viewing window of a graph is going to be exactly what you want—but not always. The most common exception is a function with vertical asymptotes. Here we are going to discuss how to adjust the viewing window.

Consider the plot of x^3 from $x = 5$ to $x = 10$, which is given by

```
plot(x^3, 5, 10)
```

and as you can imagine, the function goes from $5^3 = 125$ up to $10^3 = 1{,}000$—

thus the origin should appear very far below the graph. This is the plot:

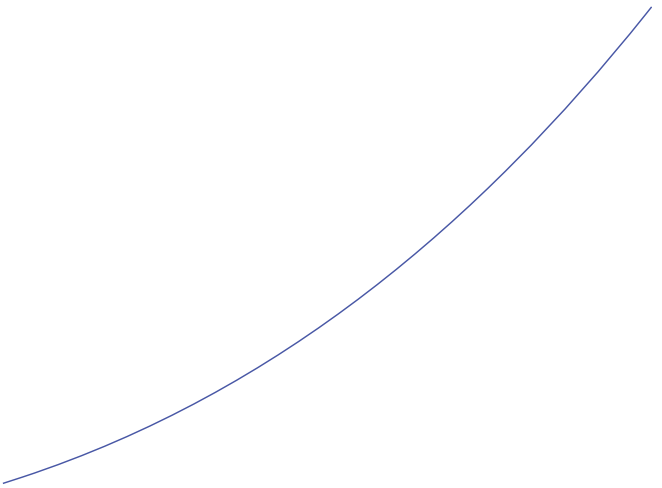


Your hint that the location of the x-axis in the display is not where it would be normally is that the axes do not intersect. This is to tell you that the origin is far away. When the axes do intersect on the screen, then the origin is (both in truth and on the screen) where they intersect.

Also, if you want to, you can hide the axes with

```
plot(x^3, 5, 10, axes=False)
```

and that produces

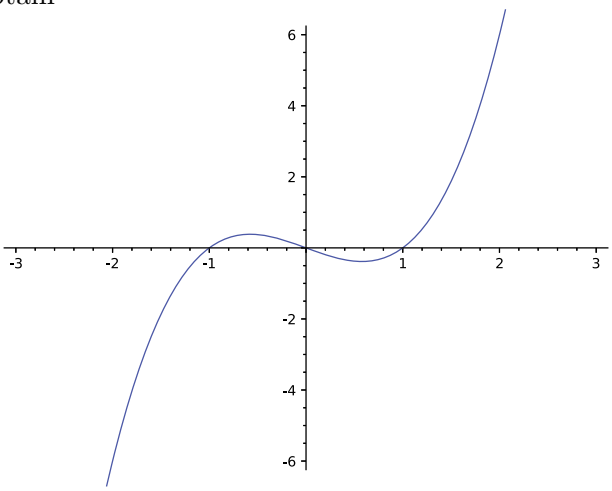


which is considerably less informative.

If, for some reason, you want to force the y-range of a graph to be constrained between two values, you can do that. For example, to keep $-6 < y < 6$, we can do

```
plot( x^3-x, -3, 3, ymin = -6, ymax = 6)
```

in order to obtain



This is important because normally Sage wants to show you the entire graph. Therefore, it will make sure that the y-axis is tall enough to include every point, from the maximum to the minimum. For some functions, either the maximum or the minimum or both could be huge.

1.4.4. Plotting Cube Roots

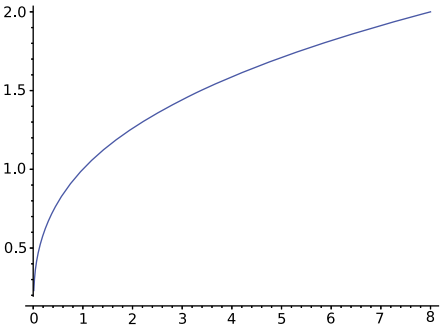
Readers who are unfamiliar with complex/imaginary numbers should skip to the next subsection. I mentioned back on page 15 in Subsection 1.2.11 that for very good reasons, Sage will return an imaginary root when you ask for the one-third power of a negative number. (Those reasons will be explained in Subsection 4.26.5, starting on page 309.)

Suppose you try to plot the cube root function with `x^(1/3)`. For negative values of x, Sage computes a complex number for `x^(1/3)`, the y-value. Since only real values of y have a place on the y-axis of an ordinary graph of any function, those points with complex y-values are not visible at all. In other words, if $y = a + bi$, where $b \neq 0$ and $i = \sqrt{-1}$, then the point (x, y) has no place in the ordinary coordinate plane. (Of course, that is also true if $x = a + bi$, where $b \neq 0$ and $i = \sqrt{-1}$.)

This means that nothing is shown for any x-value where $x < 0$, when plotting $x^{1/3}$. To see that for yourself, try the command

```
plot( x^(1/3), x, -8, 8 )
```

which produces the output



accompanied with the following error messages:

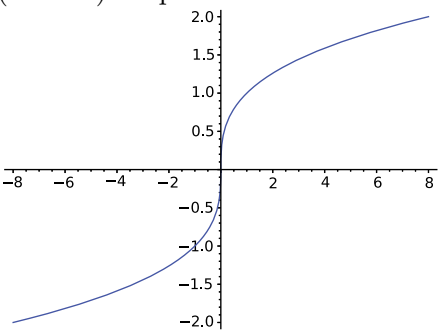
```
verbose 0 (3839: plot.py, generate_plot_points)
WARNING: When plotting, failed to evaluate function at 100 points.
verbose 0 (3839: plot.py, generate_plot_points)
Last error message: 'can't convert complex to float'
```

Of course, this graph that we just saw does not show anything for $x < 0$, which means that this plot is not correct mathematically.

To circumvent this difficulty, I lobbied for a new command to be added to Sage, and it was decided that it should be called `real_nth_root`. This command has been available in Sage since Version 9.2. As a result, we can now plot the cube-root function properly, using the code

```
plot( real_nth_root( x, 3), x, -8, 8 )
```

which produces the (correct) output



1.4.5. Plots of Functions with Vertical Asymptotes

The way that Sage (and all computer algebra tools) computes the plot of a function is by generating a very large number of points in the interval of the x's and evaluating the function at each of those points. To be precise, if you want to graph $f(x) = 1/x^2$ between $x = -4$ and $x = 4$, the computer might pick 10,000 random values of x between -4 and 4, find the y-values by plugging them into $f(x)$, and then finally drawing the dots in the appropriate spots on the graph.

So if the graph has a vertical asymptote, then near that asymptote, the value will be huge. Because of this, when you graph a rational function, be sure to restrict the y-values. For example, compare the following:

Plot 1:

```
plot(1/(x^3-x), -2, 2)
```

Plot 2:

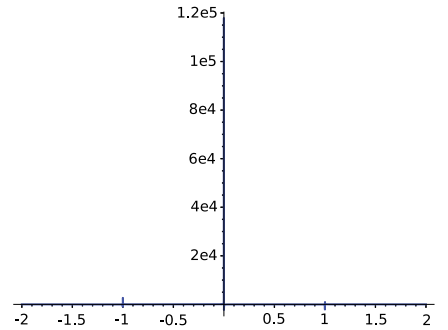
```
plot(1/(x^3-x), -2, 2,  ymin = -5,  ymax = 5)
```

Plot 3:

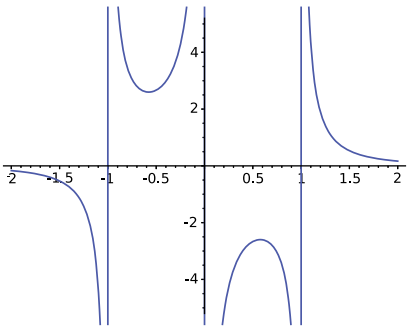
```
plot(1/(x^3-x), (x,-2, 2),  detect_poles=True,
          ymin = -5,  ymax = 5)
```

Plot 4:

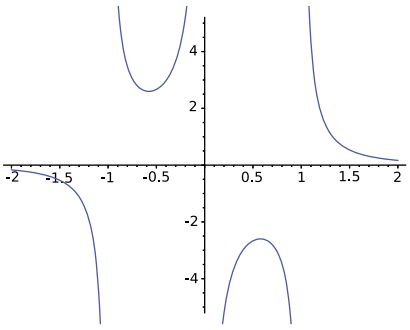
```
plot(1/(x^3-x), (x,-2, 2),  detect_poles='show',
          ymin = -5,  ymax = 5)
```



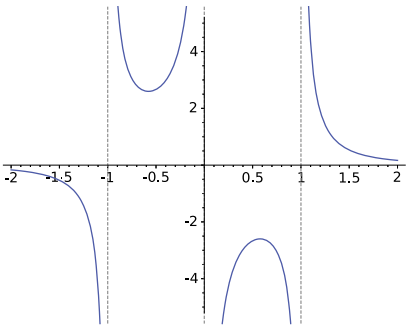
Plot 1



Plot 2



Plot 3



Plot 4

As you can see, the first is a disaster. The second one cuts off the very high and very low y-values, but it keeps trying to connect the various "limbs" of the graph. When you set "detect poles" to `True`, then it will figure out that the pieces are not connected.

Finally, we see in the fourth case that the same is true if we set "`detect_poles`" to `'show'` but, additionally, Sage will show the vertical asymptotes in such a way that they don't get confused with the graph of the function.

One minor point remains. Did you notice in the four plots above that `'show'` is in quotes, but `True` is not in quotes? That's because `True` and `False` occur so often in math and computer science, that they are built-in keywords in all programming languages (at least that I've ever seen), including Python. Sage is built on Python and has inherited `True` and `False` as keywords. However, `'show'` occurs less often and therefore is not built in, and we must put it in quotes.

1.4.6. An Example from Calculus: The Tangent Line

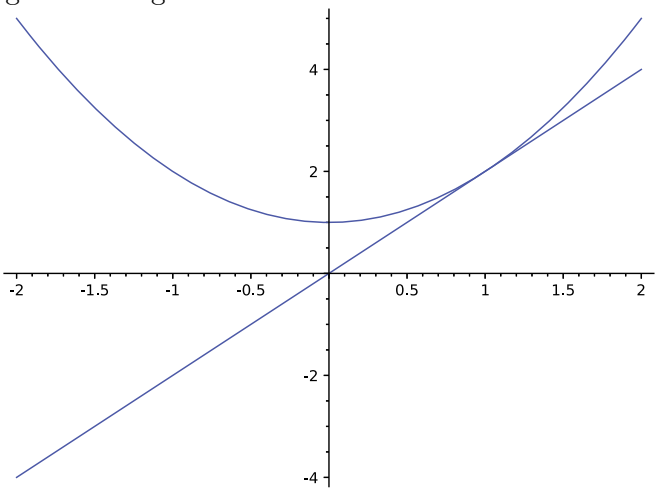
Now we're going to see how to superimpose plots on each other. It turns out that Sage thinks of this as "adding" the plots together, using a plus sign.

Suppose one wants to draw a picture of $f(x) = x^2 + 1$ over the interval $-2 < x < 2$ and the tangent line to that parabola at $x = 1$. Because $f(1) = 2$ and $f'(1) = 2$, we know the line has to go through the point $(1, 2)$ and will have slope 2. It is not hard to compute that the equation of that line is $y = 2x$. The trick is that we want to graph them both at the same time, in the same picture.

The command for this will be

```
plot( 2*x, -2, 2 ) + plot( x^2+1, -2, 2 )
```

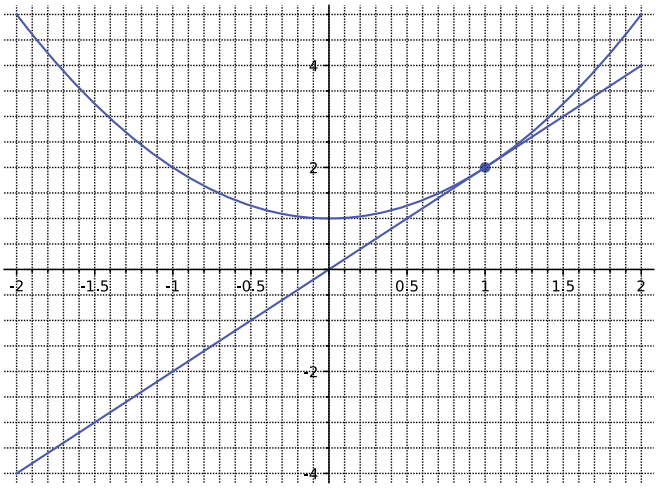
and we will get the image



As you can see, adding the two plots makes a superimposition. The plus sign tells Sage to draw the two curves on top of each other. Now, we can do better by adding a dot at the point of tangency—the point $(1, 2)$—and perhaps some gridlines. The command for that will be

```
plot( 2*x, -2, 2, gridlines='minor' ) + plot( x^2+1, -2, 2 ) +
point( (1,2), size=30 )
```

and we will get the image



As you can see, we added the point using the `point` command and this too was superimposed using the plus sign. The `gridlines='minor'` gave us the very satisfactory grid which can be so useful in preparing nice graphs that are readable, for lab reports, classroom use, or papers for publication. We will discuss other ways of making gridlines, as well as other ways of annotating a graph, in Subsection 3.1.2, starting on page 156. This specific graph will appear in a highly annotated way as Figure 3.1.

By the way, it is important that the above command, adding the two plots plus a point, must all be on the same line. One cannot have a line break. I cannot typeset it on one line because the page I am typing on is not infinitely wide. Be sure there are no line breaks when you type those commands in.

1.4.7. Superimposing Many Graphs in One Plot

Suppose you were investigating polynomials and you wanted to know the effect of varying the last numeral in

$$y = (x-1)(x-2)(x-5)$$

on its graph. (In other words, you want to know what happens if you change the "5" in that equation to other numbers.) Then you could consider looking at $(x-3)$ and $(x-4)$ as replacements for the $(x-5)$, as well as $(x-6)$ and $(x-7)$.

Maybe we might consider the domain $-2 < x < 10$. If so, then one way that this could be done[9] is by using

```
plot((x-1)*(x-2)*(x-3), -2, 10) + plot((x-1)*(x-2)*(x-4), -2,
10) + plot((x-1)*(x-2)*(x-5), -2,10) + plot((x-1)*(x-2)*(x-6),
-2, 10) + plot((x-1)*(x-2)*(x-7), -2, 10)
```

[9]By the way, when you enter that huge jumble of code into Sage, it is important to note that it must be on one (very long) line.

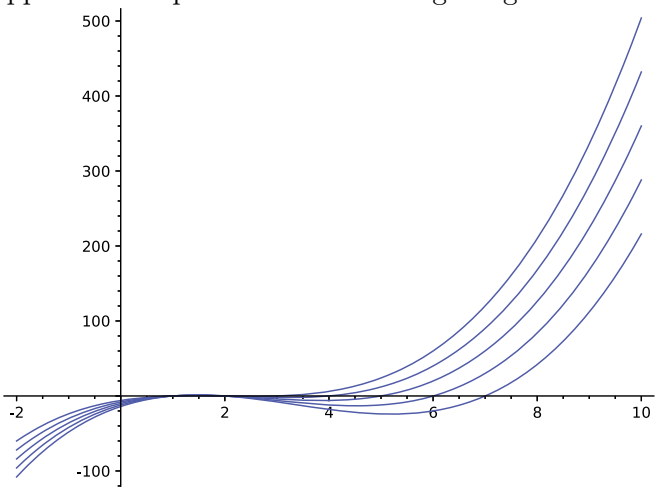
However, that code is just not human readable. Who can make sense of such a jumble of letters, symbols, and numerals?! Instead, we can type

```
P1 = plot((x-1)*(x-2)*(x-3), -2, 10)
P2 = plot((x-1)*(x-2)*(x-4), -2, 10)
P3 = plot((x-1)*(x-2)*(x-5), -2, 10)
P4 = plot((x-1)*(x-2)*(x-6), -2, 10)
P5 = plot((x-1)*(x-2)*(x-7), -2, 10)

big_plot = P1 + P2 + P3 + P4 + P5

show(big_plot)
```

That's much more readable, and we see that the plus sign (+) is again used with plots to indicate superimposition. The `big_plot` is our name for the superimposition of five plots, which we have named `P1`, `P2`, `P3`, `P4`, and `P5`. Either approach will produce the following image:



It seems that I should zoom in a bit, to see more detail. If I wanted to change the domain from $-2 < x < 10$ to perhaps $1 < x < 6$, I would have to change each of the -2's to 1's and each of the 10's to 6's. So I could type

```
plot((x-1)*(x-2)*(x-3), 1,6) + plot((x-1)*(x-2)*(x-4), 1,
6) + plot((x-1)*(x-2)*(x-5), 1,6) + plot((x-1)*(x-2)*(x-6), 1,
6) + plot((x-1)*(x-2)*(x-7), 1,6)
```

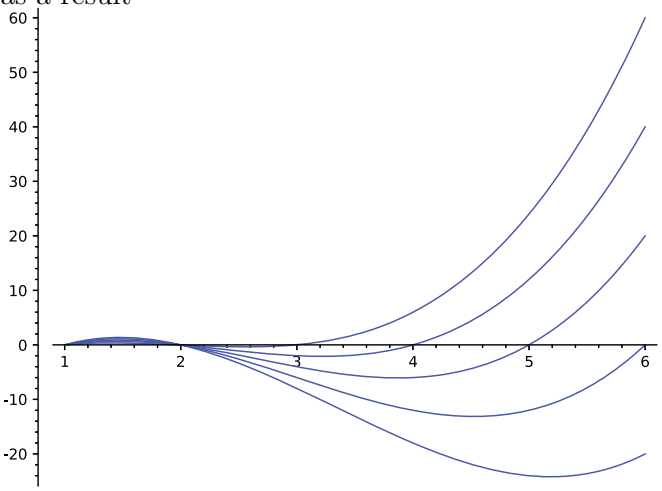
or much preferably I could type

```
P1 = plot((x-1)*(x-2)*(x-3), 1, 6)
P2 = plot((x-1)*(x-2)*(x-4), 1, 6)
P3 = plot((x-1)*(x-2)*(x-5), 1, 6)
P4 = plot((x-1)*(x-2)*(x-6), 1, 6)
P5 = plot((x-1)*(x-2)*(x-7), 1, 6)

big_plot = P1 + P2 + P3 + P4 + P5

show(big_plot)
```

and obtain as a result



There is an important issue here. We have to be careful to make the changes correctly, and that somewhat challenging requirement is made much worse when the code is hard to read. Making the code more readable also makes the code easier to change.

In any case, now I can see better what that last numeral does. It controls the location, or x-coordinate, of the last time that the curve crosses the x-axis.

By the way, you cannot do

```
plot( (x-1)*(x-2)*(x-3), 1, 6)
+ plot( (x-1)*(x-2)*(x-4), 1, 6)
+ plot( (x-1)*(x-2)*(x-5), 1, 6)
+ plot( (x-1)*(x-2)*(x-6), 1, 6)
+ plot( (x-1)*(x-2)*(x-7), 1, 6)
```

which Sage thinks is five separate lines of code, instead of one big line of code. There is a technicality here. You must not break a new line among the plots and plus signs. Generally, you can always put a new line between two completely self-contained instructions. Sometimes, you can insert a new line elsewhere, but not always. You will naturally get accustomed to how that works as you proceed through this book.

1.4.8. The Backslash Operator for Very Long Lines

There is yet another way to enter the code that we just wrote.

```
plot( (x-1)*(x-2)*(x-3), 1, 6) + plot( (x-1)*(x-2)*(x-4), 1, 6) + \
plot( (x-1)*(x-2)*(x-5), 1, 6) + plot( (x-1)*(x-2)*(x-6), 1, 6) + \
plot( (x-1)*(x-2)*(x-7), 1, 6)
```

Here, the backslash operator (\) is used to glue together those three lines into one huge line. It is important not to confuse the backslash with the forward slash (/), sometimes called a solidus, which is meant to indicate

division. As I mentioned back on page 1 in Subsection 1.1.1, on keyboards in the USA, the forward slash is located with the question mark (`?`), while the backslash is located with the vertical line (`|`), sometimes called a pipe.

If (for some reason) you might desire to have one very long line of code, or if it is unavoidable, then the backslash operator gives you a way to make that work. However, this is considered poor programming style and it is rarely necessary. I personally avoid the backslash operator.

1.4.9. Further Graphing and Plotting Topics

We have now only scratched the surface of plotting in Sage. There is a rich library of types of plots and graphs which Sage can produce for you. An entire chapter of this book is dedicated to "Advanced Plotting Techniques" (Chapter 3, starting on page 155). If you're curious about how Sage actually generates the images, we talk about that in Subsection 5.7.1, starting on page 381, but you might have to read most of Chapter 5 to be able to follow that discussion.

By the way, you do not need to read Chapter 3 in order. The sections can be read independently, as needed. You can simply use the index or table of contents to select exactly which type of graph or plot you would like and just read only the matching subsection of that chapter.

1.4.10. A Challenge: Graphing Three Unusual Functions

Using Sage, graph the following functions for $-3 < x < 3$, superimposed on the same plot. Verify that your plot is correct by mentally plugging in a few numbers.

$$\begin{aligned} f(x) &= x - 3 + |x|, \\ g(x) &= x - 2 + |x|, \\ h(x) &= x - 1 + |x|. \end{aligned}$$

1.5. Matrices and Sage, Part 1

In this section we'll learn how to solve linear systems of equations using matrices in Sage. This section assumes that you've never worked with matrices before in any way, or alternatively, that you have forgotten them. Experts in matrix algebra can skip to Subsection 1.5.3, starting on page 36. On the other hand, some readers will have no specific interest in matrices and can therefore skip to Section 1.6, starting on page 45, where they will learn how to define their own functions in Sage.

1.5.1. A First Taste of Matrices

Let us suppose that you wish to solve this linear system of equations:

$$\begin{aligned} 3x - 4y + 5z &= 14, \\ x + y - 8z &= -5, \\ 2x + y + z &= 7. \end{aligned}$$

First you would convert those equations into the following matrix:

$$A = \left[\begin{array}{ccc|c} 3 & -4 & 5 & 14 \\ 1 & 1 & -8 & -5 \\ 2 & 1 & 1 & 7 \end{array}\right].$$

Notice that the coefficients of the x's all appear in the first column; the coefficients of the y's all appear in the second column; the coefficients of the z's all appear in the third column. The fourth column gets the constants. Thus we have encapsulated and abbreviated all of the information of the problem. Furthermore, observe that additions are represented by a positive coefficient, and subtractions by a negative coefficient. By the way, the vertical line between the third and fourth columns is just decorative—its purpose is to show that the fourth column is "special" in that it contains constants (numbers), whereas the other columns represent the coefficients of the variables x, y, and z.

Matrices have a special form called "Reduced Row Echelon Form" often abbreviated as RREF. The RREF is, from a certain perspective, the simplest description of the system of equations that is still true. A formal definition of the RREF can be found in any textbook about linear algebra (the mathematics of matrices). The Reduced Row Echelon Form of this matrix A is

$$\left[\begin{array}{ccc|c} 1 & 0 & 0 & 3 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{array}\right].$$

We can translate this literally as

$$\begin{aligned} 1x + 0y + 0z &= 3, \\ 0x + 1y + 0z &= 0, \\ 0x + 0y + 1z &= 1, \end{aligned}$$

or in simpler notation

$$x = 3, \qquad y = 0, \qquad z = 1,$$

which is the solution. You should take a moment now, plug these three values into the original system of equations, and see that it comes out correctly.

1.5.2. Complications in Converting Linear Systems

You might be looking at the previous discussion and imagine that what we're doing is a kind of silver bullet, quickly capable of solving any of a large number of extremely tedious problems that take (with a pencil) a very long time.

Indeed, matrix algebra (also called linear algebra) is a silver bullet, now that we have computers. Prior to the computer, large linear algebra problems were considered extremely unpleasant. Yet, now that we have computers, and because computers can carry out these operations essentially instantly, this topic is a great way to rapidly solve enormous systems of linear equations. In turn, skilled mathematicians can address important problems in science and industry because real-world problems often have many variables.

This topic, however, is better described as "semi-automatic" rather than "automatic." The reason I say that is because the human must be very careful to first convert the word problem into a system of linear equations. We will not cover that here. The second step is to convert that system into a matrix, and that isn't as easy as it looks. In fact, it is often the case that student errors occur in this step, so we will invest a bit of time with a detailed example.

Consider this system of equations:

$$\begin{aligned} 2x - 5z + y &= 6 + w, \\ 5 + z - y &= 0, \\ w + 3(x + y) &= z, \\ 1 + 2x - y &= w - 3x. \end{aligned}$$

As it turns out, this system of equations has the following traps:

- In the first equation, the variables are out of order, which is extremely common. Very often, both students and professional mathematicians will fail to notice this and will put the wrong coefficients in the wrong places. You can use any ordering that you want, so long as you use the same ordering for every equation. However, to avoid making an error, mathematicians usually put the variables in alphabetical order.
- Also in the first equation, the w is on the wrong side, and we must move it to the left of the equal sign. It is necessary that all the variables end up on one side of the equal sign and all the constants on the other side of the equal sign. With these two repairs, the first equation is now

$$-w + 2x + y - 5z = 6.$$

- The second equation has the constant on the wrong side, so we must move it across the equal sign, remembering to negate it.

- As if that were not enough, both x and w are missing in the second equation. We treat this as if the coefficients were zero. With these two repairs, the second equation is now

$$0w + 0x - y + z = -5.$$

- The third equation has parentheses—that's not allowed. We have to remember that $3(x + y) = 3x + 3y$.
- Also in the third equation, the z is on the wrong side. We must take care to change $+z$ to $-z$ when crossing the equal sign.
- A third defect in the third equation is that there is no constant. We treat this as a constant of zero. Correcting these three issues, the third equation becomes

$$w + 3x + 3y - z = 0.$$

- Now the real delinquent is the fourth equation. We have w on the wrong side, and what is worse is that there are two occasions of x. When we move the $-3x$ from the right of the equal sign to the left, it becomes $+3x$ and combines with the $+2x$ that was already there, to form $5x$. As if that were not enough, the constant is on the wrong side. Last but not least, z is entirely absent. Correcting all these defects, we have

$$-w + 5x - y + 0z = -1.$$

With these (modified) equations in mind, we have the matrix

$$B = \left[\begin{array}{cccc|c} -1 & 2 & 1 & -5 & 6 \\ 0 & 0 & -1 & 1 & -5 \\ 1 & 3 & 3 & -1 & 0 \\ -1 & 5 & -1 & 0 & -1 \end{array}\right].$$

The RREF of that matrix would be tedious to find by hand. However, it would be far worse to solve that system of equations by hand without matrices, using ordinary algebra.

In contrast, we will easily compute the RREF, using Sage, in the next subsection.

For now, it turns out that the answer is

$$w = -107/7, \qquad x = -12/7, \qquad y = 54/7, \qquad z = 19/7,$$

which you can easily verify now if you like.

1.5.3. Computing the RREF in Sage

This subsection is going to tell you how to compute the RREF of a matrix, presumably with the goal of solving some linear system of equations. Doing this in Sage requires only four steps, the last of which is optional. First,

we're going to define the matrix

$$A = \left[\begin{array}{ccc|c} 3 & -4 & 5 & 14 \\ 1 & 1 & -8 & -5 \\ 2 & 1 & 1 & 7 \end{array}\right],$$

which came from page 34 in Subsection 1.5.1. As you can see, it has 3 rows and 4 columns. We'll do that with the Sage command

```
A = matrix( 3, 4, [ 3, -4, 5, 14, 1, 1, -8, -5, 2, 1, 1, 7 ] )
```

The key to understanding that Sage command is to see that the first number is the number of rows, followed by the number of columns. How can we remember that? This is my memory hook: Columns, such as on a bank, the US Congress, or a Greek temple, are vertical. Rows, by process of elimination, are horizontal. Then follows the list of entries, separated by commas and enclosed in brackets. But do we enter the rows first or the columns first? Imagine reading words in a novel. First you read all the words in the first line, then you start with the second line, and all its words, before moving to the third line. Similarly, first you enter all the entries of the first row of a matrix, then you start with the second row and all its entries, before moving to the third row (if there is one).

By the way, keep this definition of the matrix A in the SageMathCell's window for as long as you are working with it. Do not erase it until you are done with it. Every time you hit "Evaluate," you are sending whatever is in the window to the SageMathCell server for evaluation. The server won't remember the definition of the matrix A if you remove it. In contrast, either a local installation or `CoCalc.com` (see Appendix B, starting on page 443) would have much more permanent memory.

Notice that Sage does not respond when this command is given. That's because we told it what the matrix A is but we didn't give it anything to do to A.

The second step is to verify that the matrix you typed was the matrix that you had hoped to enter. This step is not avoidable, as typos are extremely common. To do this, just type `print(A)` on a line by itself, below the definition of the matrix A. Then click "Evaluate." The matrix is displayed and you can verify that you have typed what you had wanted to type.

The third step is that you want to compute the RREF or "Reduced Row Echelon Form." This is done by replacing `print(A)` with instead

```
print(A.rref())
```

and Sage responds with

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

which is the Reduced Row Echelon Form.

Alternatively, to see a more beautiful style of output, we could have typed `show(A.rref())` instead of `print(A.rref())`. As we will see throughout this book, the `show()` command has many different uses.

The fourth step is to check your work. It is optional, but I highly recommend it! Either with a hand calculator or with Sage, verify that

$$3(3) - 4(0) + 5(1) = 14 \text{ as expected}$$

and also

$$1(3) + 1(0) - 8(1) = -5 \text{ as expected}$$

and finally

$$2(3) + 1(0) + 1(1) = 7 \text{ as expected.}$$

Note that it is crucial to check all three equations! Otherwise, you'd fail to detect the "imposter" solution

$$x = 30, \qquad y = 29, \qquad z = 8,$$

which satisfies the first two equations, but not the third one.

Now let's consider the "complicated" example from the previous subsection. It was

$$B = \left[\begin{array}{cccc|c} -1 & 2 & 1 & -5 & 6 \\ 0 & 0 & -1 & 1 & -5 \\ 1 & 3 & 3 & -1 & 0 \\ -1 & 5 & -1 & 0 & -1 \end{array}\right]$$

and we can enter that into Sage with

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

With the matrix B being built out of a very long list of numbers, we should be very careful that we've typed those correctly, omitting none and duplicating none. We check the correctness of B by typing `print(B)` alone on its own line and see that we have entered that which we had intended to enter.

Before we continue, I'd like to share with you a useful and easy bit of notation. Sometimes we wish to refer to a position inside the matrix. For example, the 6 in the first row, fifth column of B can be written as B_{15}. Likewise, the 5 in the fourth row, second column of B can be written as B_{42}. Similarly, the two -5's are located at B_{14} and B_{25}. Remember, the row comes first, and the column comes second.

In Sage, the naming of the coordinates in matrix entries starts at 0 and not at 1. This means that the upper left-hand corner is `B[0,0]` in Sage, but B_{11} in mathematics. Similarly, B_{14} is typed in as `B[0,3]`, and B_{23} is typed in as `B[1,2]`. This anomaly is something that Sage inherited from the computer language Python, and it cannot be changed since Sage runs over Python. Luckily, you will find that you do not often have to access individual entries in a matrix—it is actually rather rare.

By the way, an alternative way of entering a fairly large matrix is as follows:

```
B = matrix(  [  [-1, 2, 1, -5, 6],  [0, 0, -1, 1, -5],
        [1, 3, 3, -1, 0],  [-1, 5, -1, 0, -1 ]  ]   )
```

The above code is a list of lists. Each list of five numbers enclosed in brackets and separated by commas represents one row; then there are four such lists, separated by commas and enclosed by brackets, to represent the entire matrix. Since this format gives away the number of rows and columns, we do not have to put the "`4, 5`" to inform Sage that B has 4 rows and 5 columns. Feel free to use whichever format you might happen to feel more comfortable with.

Now that B has been entered (by either method) we should add the command `print( B.rref() )` on its own line, and we learn that the RREF is

$$\left[\begin{array}{cccc|c} 1 & 0 & 0 & 0 & -107/7 \\ 0 & 1 & 0 & 0 & -12/7 \\ 0 & 0 & 1 & 0 & 54/7 \\ 0 & 0 & 0 & 1 & 19/7 \end{array}\right],$$

which gives the final answer of

$$w = -107/7, \qquad x = -12/7, \qquad y = 54/7, \qquad z = 19/7$$

for the original system of equations. Now we must check our work. For example, we check the first equation with

```
print( 2*(-12/7) - 5*(19/7) + 54/7, 'compared to', 6 + -107/7 )
```

and we receive the output

```
-65/7 compared to -65/7
```

The point is not that they come out to $-65/7$, but rather that they come out equal. Thus the first equation is satisfied. We check the other three equations similarly.

We really do have to plug each of the four values into each of the four equations, by the way. For example, the "imposter" solution

$$w = -139/7, \qquad x = -8/7, \qquad y = 64/7, \qquad z = 29/7$$

satisfies the first three of those equations but fails to satisfy the last one.

If you happen to have read Subsection 1.5.2, starting on page 35, you will recall that we did rather a lot of work to get B. Those many steps might have contained an error if we were sloppy. That's another reason that we should check our work.

1.5.4. RREFs and the Identity Matrix

We've now learned how to use the `matrix` and `rref` commands to compute the RREF of a matrix and therefore solve a linear system of equations. To

do all the steps at once, I will often type

```
B = matrix( 4, 5, [-1, 2, 1, -5, 6, 0, 0, -1, 1, -5, 1, 3, 3,
        -1, 0, -1, 5, -1, 0, -1 ] )
print('Question:')
print(B)
print('Answer:')
print( B.rref() )
```

The lines containing `print('Question:')` and `print('Answer:')` are entirely optional but help make the output more readable. They also encourage me to check that I had actually entered the matrix which I had intended to enter, without any typos. Somehow, I almost always make a typographical error, which I have to go back and correct. The output produced is

```
Question:
[-1  2  1 -5  6]
[ 0  0 -1  1 -5]
[ 1  3  3 -1  0]
[-1  5 -1  0 -1]
Answer:
[     1      0      0      0 -107/7]
[     0      1      0      0  -12/7]
[     0      0      1      0   54/7]
[     0      0      0      1   19/7]
```

Let's look at the RREFs that we've computed so far. Do you see a pattern in them? Perhaps I should say, do you see a pattern in all columns excluding the last column in

$$\left[\begin{array}{ccc|c} 1 & 0 & 0 & 3 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{array}\right] \qquad \text{and} \qquad \left[\begin{array}{cccc|c} 1 & 0 & 0 & 0 & -107/7 \\ 0 & 1 & 0 & 0 & -12/7 \\ 0 & 0 & 1 & 0 & 54/7 \\ 0 & 0 & 0 & 1 & 19/7 \end{array}\right] ?$$

We can see that all the columns (except the last column) have zeros everywhere, except a very noticeable diagonal of ones. This pattern (of zeros everywhere but ones in the A_{11}, A_{22}, A_{33}, ... positions) is a pattern that occurs extremely often. Unsurprisingly, it has a name—mathematicians call this "the identity matrix."

Consider when you are forced to show an "ID," for example when being pulled over for a speeding ticket or when trying to enter a bar. The ID quickly displays your vital facts, such as your name, your birthdate, your height, your gender, your address, and so forth. Likewise, the identity matrix displays the vital facts about a system of linear equations. You can read the values of the variables off, by simply reading the last column.

1.5.5. A Challenge: Solving a Linear System of Equations

Using Sage, solve the following linear system of equations. Check your answers with a hand calculator.

$$\begin{aligned} 3{,}481x + 59y + z &= 0.87, \\ 6{,}241x + 79y + z &= 0.61, \\ 9{,}801x + 99y + z &= 0.42. \end{aligned}$$

Note: This problem might seem artificial, but it is actually an applied problem which we will revisit shortly. Like many application-based problems, the answers have decimal points, which hopefully should not frighten you. We'll see the answer shortly.

1.5.6. Vandermonde's Matrix

In economics or in business, one often speaks a great deal about the price-demand curve. After all, as the price of a product goes up, then the demand for it should fall, and as the price decreases, the demand should go up, at least under standard conditions.

Sometimes it would be useful to know the curve but it can usually only be computed by surveying potential customers. The standard technique is to find 2,000 people, show each of them the product, and ask them if they would buy it or not. They get the product[10] for free in return for participating in the survey. The key is that of those 2,000 people, perhaps 20 different prices would be used, each price being shown to 100 people.

That process will deliver 20 data points, which is often totally sufficient. However, sometimes it is better to have an actual function. Typically, a quadratic function is sought. There are good reasons for this, but it would be a digression[11] to explore them here. Imagine that a friend of yours with a startup company has done such a survey, but with 3 prices and 300 people, for some snazzy new product.

- At the price of \$59, the survey says that 87% would buy it.
- At the price of \$79, the survey says that 61% would buy it.
- At the price of \$99, the survey says that 42% would buy it.

Every quadratic function can be written in the form

$$f(x) = ax^2 + bx + c$$

but the question becomes: How can we find a, b, or c?

[10] Unsurprisingly, this technique is more common in small consumer electronics or food items and less common among luxury sports car manufacturers.

[11] Let it be said that fitting high-degree polynomials to data points is entirely feasible, but extremely unwise, due to a phenomenon called "overfitting."

Assuming $f(x)$ models the price-demand relationship well, then it really must be the case that $f(59) = 0.87$, $f(79) = 0.61$, and $f(99) = 0.42$. With that in mind, then we could write the following equations:

$$\begin{aligned} a(59^2) + b(59) + c &= 0.87, \\ a(79^2) + b(79) + c &= 0.61, \\ a(99^2) + b(99) + c &= 0.42. \end{aligned}$$

Surprisingly, these equations are linear, which is not necessarily expected since we are working with a quadratic function. All we must do is write the matrix down,

$$\left[\begin{array}{ccc|c} 59^2 & 59 & 1 & 0.87 \\ 79^2 & 79 & 1 & 0.61 \\ 99^2 & 99 & 1 & 0.42 \end{array}\right],$$

and ask Sage to compute the RREF. However, you've already done that yourself! You already computed the solution

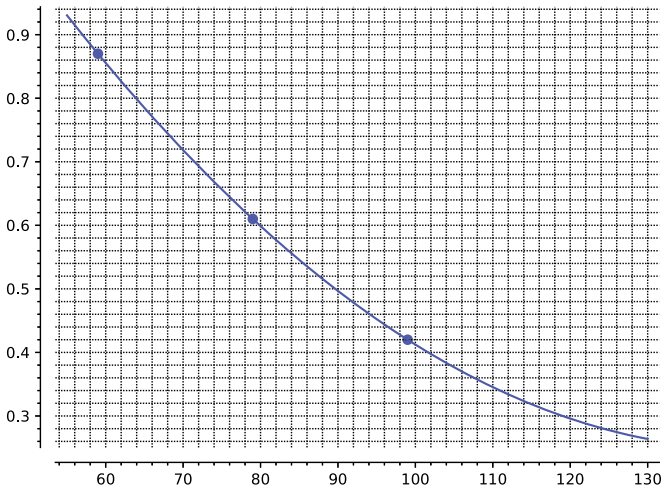
$$a = 0.0000875, \qquad b = -0.025075, \qquad c = 2.0448375$$

as practice, as a challenge on Subsection 1.5.5, starting on page 41.

Finally, we conclude that the function is

$$f(p) = 0.0000875p^2 - 0.025075p + 2.0448375.$$

The graph of this function is



If you are curious, the code which produced that plot is

```
f(x) = 0.0000875*x^2 - 0.025075*x + 2.0448375

plot( f, 55, 130, gridlines='minor') + point( [59, 0.87],
size=50 ) + point([79, 0.61], size=50) + point(
[99, 0.42], size=50 )
```

The general technique of fitting a degree-n polynomial to $n+1$ points using a matrix is due to Alexandre-Théophile Vandermonde (1735–1796), and

therefore the matrices are called Vandermonde Matrices after him. For example, to fit a fourth-degree polynomial to five data points (x_1, y_1), (x_2, y_2), $\ldots$, (x_5, y_5), we would use the matrix

$$\left[\begin{array}{ccccc|c} (x_1)^4 & (x_1)^3 & (x_1)^2 & (x_1) & 1 & y_1 \\ (x_2)^4 & (x_2)^3 & (x_2)^2 & (x_2) & 1 & y_2 \\ (x_3)^4 & (x_3)^3 & (x_3)^2 & (x_3) & 1 & y_3 \\ (x_4)^4 & (x_4)^3 & (x_4)^2 & (x_4) & 1 & y_4 \\ (x_5)^4 & (x_5)^3 & (x_5)^2 & (x_5) & 1 & y_5 \end{array}\right].$$

1.5.7. Linear Systems of Equations with No Solutions

Previously, our examples gave us one unique solution. Now we're going to examine the two semi-rare cases. These occur if there is one row of all zeros at the bottom of the RREF, either ending in a non-zero number or ending in zero. Consider the following system of equations:

$$\begin{aligned} x + 2y + 3z &= 7, \\ 4x + 5y + 6z &= 16, \\ 7x + 8y + 9z &= 24. \end{aligned}$$

This results in the matrix

$$C_1 = \left[\begin{array}{ccc|c} 1 & 2 & 3 & 7 \\ 4 & 5 & 6 & 16 \\ 7 & 8 & 9 & 24 \end{array}\right]$$

and that we shall enter into Sage with

```
C1 = matrix(3,4, [1, 2, 3, 7, 4, 5, 6, 16, 7, 8, 9, 24])
```

Next, to find the RREF we type

```
C1.rref()
```

and receive back

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

This translates into

$$\begin{aligned} x - z &= 0, \\ y + 2z &= 0, \\ 0 &= 1. \end{aligned}$$

The bottom equation says $0 = 1$, which is clearly not true! There is no way to satisfy the requirement that $0 = 1$. Therefore, this equation has no solutions. Instead of "the answer" being a solution, a list of solutions, or infinitely many solutions, the answer is the sentence "This linear system of equations has no solutions." That will be the outcome whenever the RREF has a row with all zeros except the last column, which is a non-zero number.

Whenever you see a row with all zeros, except in the last column where a number other than zero is found, then you must remember that the system has no solutions.

1.5.8. Linear Systems with Infinitely Many Solutions

Another, but different, case is to change the 24 in the last equation to 25. That results in the matrix

$$C_2 = \left[\begin{array}{ccc|c} 1 & 2 & 3 & 7 \\ 4 & 5 & 6 & 16 \\ 7 & 8 & 9 & 25 \end{array}\right]$$

and that we shall enter into Sage with

```
C2 = matrix(3,4, [1, 2, 3, 7, 4, 5, 6, 16, 7, 8, 9, 25])
```

Now, to find the "solution," we type

```
C2.rref()
```

and receive back

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

As you can see, this RREF has a row of all zeros, ending in a zero. That usually indicates infinitely many solutions, a point we will clarify shortly. Some instructors allow you to write "infinitely many solutions" and in certain application problems you would not care to provide a way of saying what those solutions are.

However, some of the time, you'd like to know what the solutions are. This is especially true in geometry-related problems. In any case, you can't make a list because there are infinitely many solutions. Often, even though the solution space is infinite, it turns out that it has what mathematicians call "one degree of freedom." This means that there's a free variable, which you can set to whatever value that you want. However, once you have done so, all the other variables are determined by this choice. Less often, there will be multiple degrees of freedom, thus multiple free variables.

So what does a row of zeros really tell us? Most of the time, your system of equations will have two equations in two unknowns; three equations in three unknowns; four equations in four unknowns; or five equations in five unknowns; and so forth. The technical term for this is an "exactly defined" linear system. An exactly defined linear system results in a 2×3, 3×4, 4×5, 5×6, or $n \times (n + 1)$ matrix. The "rule of thumb" that I'm about to present will work whenever there are an equal number of equations as unknowns, which means that the system is exactly defined, i.e., there is one more column than the number of rows.

Here's the rule of thumb: If we see the tell-tale sign of "no solutions," then we ignore all other information and declare "no solutions." Barring that, if there is a row of zeros, then there will be infinitely many solutions.

However, be warned! This rule does not work if there are "too few" or "too many" equations. If there are too many equations (too many rows), the system is said to be "overdefined." If there are too many variables (too many columns), the system is said to be "underdefined." This rule of thumb is false for matrices that are overdefined or underdefined. If you are curious, underdefined systems of linear equations come up in some cool analytic geometry questions, and overdefined systems of linear equations come up in cryptanalysis—and were used in my PhD dissertation.

1.6. Making Your Own Functions in Sage

It is extremely easy to define your own functions[12] in Sage. To me, this is one of the most brilliant design features of Sage, that the creators made functions very easy to use. You can make your own function using exactly the same symbols that you would normally use in writing the function down with your pencil on homework or writing on the whiteboard.

1.6.1. Defining a Function

For example, if you want to define $f(x) = x^3 - x$, then you type

```
f(x) = x^3-x
f(2)
```

where we have defined $f(x)$ and asked it the value of $f(2)$. You could instead ask for $f(3)$ or any other number. Likewise if you wanted to define $g(x) = \sqrt{1-x^2}$, you would type

```
g(x) = sqrt( 1-x^2 )
g(1/4)
```

where we have defined $g(x)$ and asked what $g(1/4)$ evaluates to. You could just as easily ask for $g(1/2)$ or any other number in the domain of g.

To evaluate several values, you can use the `print` command, as we've seen before.

```
g(x) = sqrt( 1-x^2 )
print( g(0.1) )
print( g(0.01) )
print( g(0.001) )
print( g(0.0001) )
print( g(0.00001) )
print( g(0.000001) )
```

[12] Just to clarify, I am talking about mathematical functions, such as $f(x) = x^3$. However, if you are an experienced computer programmer, you might know about defining functions in a programming language—sometimes called procedures, methods, or subroutines. That will be discussed in Section 5.2, starting on page 331.

If perhaps you happen to have started calculus, you can see that the above output would help you evaluate

$$\lim_{x \to 0^+} g(x) = 1$$

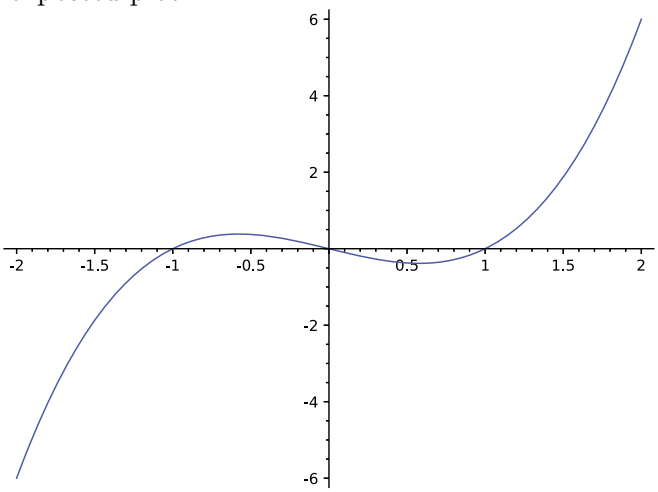
numerically, or help confirm your algebraic answer. We will discuss this further in Subsection 5.1.6, starting on page 324. If you haven't studied calculus, then you can ignore this remark!

1.6.2. Plotting Functions

Now that we've defined $f(x)$ and $g(x)$, we can plot them with

```
f(x) = x^3-x
plot( f, -2, 2 )
```

You get the expected plot:



Likewise if you type

```
g(x) = sqrt( 1-x^2 )
plot( g, 0, 1 )
```

which is equivalent to

```
g(x) = sqrt( 1-x^2 )
plot( g(x), 0, 1 )
```

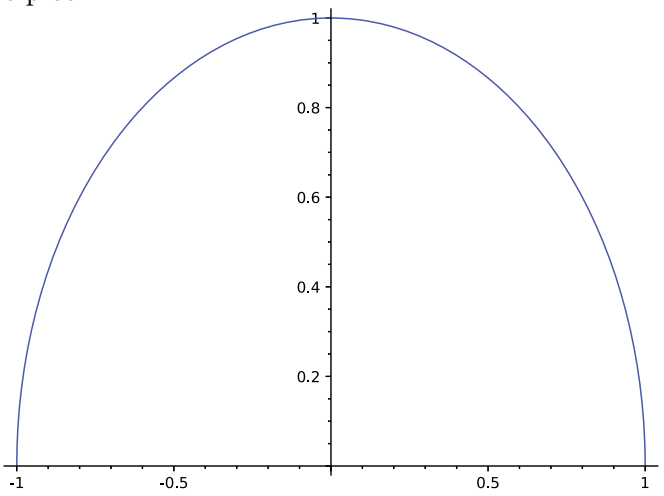
then you get the graph of $g(x)$ on $0 \leq x \leq 1$.

Last but not least, if you type

```
g(x) = sqrt( 1-x^2 )
plot( g )
```

which is rather abbreviated—omitting the interval of the x-axis that you want—then Sage will assume that you want $-1 \leq x \leq 1$, by default. This

produces the plot



Note that you really do have to keep the definitions of $f(x)$ or $g(x)$ in the window and written out each time you hit "Evaluate." This is a feature of the SageMathCell server. It has amnesia and treats each press of the button "Evaluate" as its own separate mathematical universe. This is for the beginner's benefit because it makes the correcting of minor typographical errors very easy. You just fix the error, click "Evaluate," and all is well. As it turns out, `CoCalc.com` works just slightly differently, as will be explained in Appendix B, starting on page 443.

1.6.3. Using Intermediate Variables

Sometimes you'll be using the same value a lot, and you'll want to store it somewhere. For example, if you type 355/113 a lot, then you can type

```
k=355/113
```

which silently stores the value 355/113 in the variable k. You can use it in any later line of the SageMathCell window, such as

```
k=355/113
print( 2+k )
print( 1+k )
```

which displays the correct values of 581/113 and 468/113. If you go back and change the value of k—however, you have to make sure you use the "Evaluate" button to re-evaluate your formulas.

Also, if you replace `2+k` with `2.0+k`, then you get decimal approximations instead. If a mathematical statement is written entirely in terms of integers and rational numbers, Sage will provide an exact answer. If a decimal is introduced, it will provide a decimal approximation instead.

The intermediate variables technique is great for things in physics, like the acceleration due to gravity or the speed of light, which do not change during a problem.

As it turns out, this particular k is an approximation of π, kind of like 22/7 but much better. If we wanted to find the relative error of k as an approximation of π, we would remember the formula

$$\text{relative error} = \frac{\text{approximation} - \text{truth}}{\text{truth}}$$

and type

```
k=355/113
N(  (k-pi)/pi  )
```

and learn that the relative error is around 84.9 parts per billion. Not bad. This approximation was found by the Chinese astronomer Zu Chongzhi, also known as Tsu Ch'ung-Chih. We can compare 355/113 to 22/7 with

```
N(   (22/7-pi)/pi   )
```

and learn that this much more common approximation of π as 22/7 has a relative error around 402.5 parts per million. That's much worse! In fact, 22/7 is 4,740.9 times worse, so it is kind of a pity that we do not teach students 355/113, but instead we teach them 22/7.

1.6.4. Comparing the Plots of Two (or More) Functions

Sometimes we might want to understand how slightly changing a function's formula will change the function. We might want to do that by seeing how that small change to the formula impacts the plot of the function. One option is to plot both functions on the same graph (as we saw in Subsection 1.4.7, starting on page 31), but there is another option. We might want the two (or more) plots to appear separately.

The following code accomplishes that task. It will display the three plots one after another (vertically), instead of superimposed on the same graph.

```
f(x) = (x-1)^2 - 1
g(x) = (x-2)^2 - 1
h(x) = (x-3)^2 - 1
P1 = plot( f(x), -4, 4 )
P2 = plot( g(x), -4, 4 )
P3 = plot( h(x), -4, 4 )
show(P1)
show(P2)
show(P3)
```

1.6.5. Composition of Functions

You can also compose two functions very easily in Sage. Consider the following two functions:

$$f(x) = 3x + 5 \qquad \text{and} \qquad g(x) = x^3 + 1.$$

Perhaps we wish to explore $f(g(x))$. The easy way to do this is to define a third function, $h(x) = f(g(x))$. Then perhaps we might want to print that

function and the values of $h(x)$ at $x = 1$, $x = 2$, $x = 3$, and $x = 4$. The Sage code for this is straightforward:

```
f(x) = 3*x + 5
g(x) = x^3 + 1
h(x) = f(g(x))
print( h(x) )
print( h(1), h(2), h(3), h(4), sep='...' )
```

That produces the output

```
3*x^3 + 8
11...32...89...200
```

as desired, where the dots have been added to keep those outputs separated, for the ease of human reading. (I will explain the `sep='...'` momentarily.)

However, if we replace the third line with $h(x) = g(f(x))$, then we get completely different output

```
(3*x + 5)^3 + 1
513...1332...2745...4914
```

Of course, there's no reason to expect $f(g(x)) = g(f(x))$ in general. The way to see that in this specific case is to work out the two functions using algebra. We can compute

$$\begin{aligned} f(g(x)) &= 3(g(x)) + 5 \\ &= 3(x^3 + 1) + 5 \\ &= 3x^3 + 3 + 5 \\ &= 3x^3 + 8 \end{aligned}$$

which is completely different than

$$\begin{aligned} g(f(x)) &= g(3x + 5) \\ &= (3x + 5)^3 + 1 \\ &= 27x^3 + 135x^2 + 225x + 125 + 1 \\ &= 27x^3 + 135x^2 + 225x + 126. \end{aligned}$$

Now let's discuss what the `sep='...'` option does. It changes the separator for that `print` statement, in the following sense. If the `sep` option is absent, then by default a space is placed between each item being printed by that particular `print` statement. Since we changed the separator to three periods (dots) and no spaces, then each number is separated by three dots and no spaces.

1.6.6. Multivariate Functions

This section has dealt with functions of one variable. Sage can very easily and simply handle functions of multiple variables. For example,

$$f(x, y) = (x^4 - 5x^2 + 4 + y^2)^2.$$

We'll cover this in detail in Section 4.1, starting on page 193.

1.6.7. A Challenge: Predicting the Behavior of Functions

In this challenge, you will consider the task of comparing the plots of

$$\sqrt{\quad f(x)^2}$$

with the plots of $f(x)$, for many functions.

First, I'd like you to think about the problem abstractly, based on what you know of mathematics, without consulting Sage. In what ways, if any, will $\sqrt{\quad f(x)^2}$ be different from the plot of $f(x)$? Think about it for a minute, and commit to one of the following choices—without using Sage until you've committed to your choice. Remember, we want to say something that is true in general, considering many classes of functions.

(a) The plot of $\sqrt{\quad f(x)^2}$ will show those parts of the plot of $f(x)$ where $f(x)$ is positive, but the parts of the plot of $f(x)$ where $f(x)$ is negative will not be shown.
(b) The plot of $\sqrt{\quad f(x)^2}$ will be identical to the plot of $f(x)$.
(c) The plot of $\sqrt{\quad f(x)^2}$ will be the plot of $f(x)$ reflected around the x-axis.
(d) The plot of $\sqrt{\quad f(x)^2}$ will be the plot of $|f(x)|$.
(e) The plot of $\sqrt{\quad f(x)^2}$ will be totally unrelated to the plot of $f(x)$, when considering many different functions
(f) The plot of $\sqrt{\quad f(x)^2}$ will be the plot of $f(x)$ reflected around the y-axis.
(g) None of these.

After you have committed to your choice, having thought about this question mathematically, then and only then is it time to consult Sage. Use the techniques of Subsection 1.6.4, starting on page 48 to compare the plots of $\sqrt{\quad f(x)^2}$ to the plots of $f(x)$, for some famous functions whose plots you'd recognize, such as $f(x) = x^2$, $f(x) = x^3 - x$, $f(x) = x$, $f(x) = \log x$, $f(x) = |x| - 1$, $f(x) = 2 - x$, and if you've studied trigonometry, $f(x) = \sin x$ and $f(x) = \cos x$. It is sufficient to consider $-2 < x < 2$, except for $\log x$, where you might want to consider $0.1 < x < 4$.

1.7. Using Sage to Manipulate Polynomials

Now we're going to explore some applications of functional notation, from the previous section, to work with polynomials. First, let's see that Sage

can add polynomials very easily. We're going to explore the following polynomials:

$$\begin{aligned} a(x) &= x^2 - 5x + 6, \\ b(x) &= x^2 - 8x + 15. \end{aligned}$$

If we type the code

```
a(x) = x^2 - 5*x + 6
b(x) = x^2 - 8*x + 15

a(x) + b(x)
```

we get the correct answer of $2x^2 - 13x + 21$. Note that it is very important to remember the asterisk between the 5 and the x as well as between the 8 and the x.

Similarly, if you change the `+` sign between `a(x)` and `b(x)` into a `-` sign, then we also get the correct answer of $3x - 9$. As you can imagine, the product can be conveniently calculated as well:

```
a(x) = x^2 - 5*x + 6
b(x) = x^2 - 8*x + 15

a(x)*b(x)
```

but this provides the unfinished yet undeniably true answer

```
(x^2 - 5*x + 6)*(x^2 - 8*x + 15)
```

Instead, we can do the following:

```
a(x) = x^2 - 5*x + 6
b(x) = x^2 - 8*x + 15

g(x) = a(x)*b(x)
g(x).expand()
```

We get $g(x)$ written without parentheses. In computer algebra, that's called "expanded form." The expanded form of our $g(x)$ happens to be

```
x^4 - 13*x^3 + 61*x^2 - 123*x + 90
```

Instead, we can write `g(x).factor()` in place of `g(x).expand()` in which case we get the output

```
(x - 2)*(x - 3)^2*(x - 5)
```

which is the "factored form" of the polynomial $g(x)$.

You can also factor more directly with

```
a(x) = x^2 - 5*x + 6
factor( a(x) )
```

and get the correct answer of $(x - 2)(x - 3)$. Alternatively we can be even more direct by typing

```
factor( x^2 - 8*x + 15 )
```

to obtain $(x - 3)(x - 5)$.

1.7.1. Computing the gcd of Two Polynomials

You can also compute the "greatest common divisor" or `gcd` of two polynomials. For example, the code

```
a(x) = x^2 - 5*x + 6
b(x) = x^2 - 8*x + 15

gcd( a(x), b(x) )
```

provides the correct answer of $(x-3)$. Of course, we knew that from the factorizations that we found earlier. The common factor of $(x-2)(x-3)$ and $(x-3)(x-5)$ is clearly $(x-3)$.

It should be noted that these examples were trivial because we used quadratic polynomials. The following polynomial is one that I would not want to factor by hand—though, admittedly, the integer roots theorem[13] would produce the correct answer using only a pencil and a great deal of human patience:

```
factor( x^4 - 60*x^3 + 1_330*x^2 - 12_900*x + 46_189 )
```

Speaking of the integer roots theorem, we could type `factor(46_189)` and get `11 * 13 * 17 * 19` to help us find roots of that quartic polynomial. The `factor` command deals with both factoring a polynomial and factoring an integer. The command `divisors(46_189)`, as an alternative, lists all the positive integers which divide 46,189. Naturally, that's a longer list than `factor`, which only would list the prime divisors (and their exponents, if any). Compare `factor(2_048)` and `divisors(2_048)`, if you like.

The composition of functions is also not a problem. The code given below will compose two polynomials and produce the answer in three forms. It is noteworthy that this is our first real "program" in Sage, in the sense that it is the first time that we've used more than two or three commands at once.

```
a(x) = x^2 - 5*x + 6
b(x) = x^2 - 8*x + 15
f(x) = a(b(x))

print('Direct:', f(x) )
print('Expanded:', f(x).expand() )
print('Factored:', f(x).factor() )
```

[13] If $f(x)$ is a polynomial with all integer coefficients, then any integer root must be a divisor of the constant term of $f(x)$. Most numbers have a modest number of divisors, so it is relatively easy to simply check each of them and thereby have a complete list of all the integer roots of $f(x)$. Note that this works even for fifth and higher-degree polynomials, which are otherwise extremely difficult to work with, as will be described in Subsection 1.9.4, starting on page 68. There is also a "rational roots theorem," but we will not discuss that here.

In particular, our program produces the output

```
Direct: (x^2 - 8*x + 15)^2 - 5*x^2 + 40*x - 69
Expanded: x^4 - 16*x^3 + 89*x^2 - 200*x + 156
Factored: (x^2 - 8*x + 13)*(x - 2)*(x - 6)
```

The above code demonstrates how it can be very useful to label parts of your output with `print` commands if you're outputting more than one or two items. Such labeling (sometimes called "annotating the output") is a good general practice as you learn to tackle more and more complex tasks in Sage.

Just as we saw on page 49 in Subsection 1.6.5, you get a different answer if you change `f(x) = a(b(x))` into `f(x) = b(a(x))`. Go ahead, give it a try, and see for yourself!

1.7.2. A Challenge: Exploring the gcd of Polynomials

For this challenge, you will be using the following polynomials. There is no need to retype them. Remember, every block of code that is longer than four lines has been included in an archive on the book's website:

`http://www.sage-for-undergraduates.org/`

so that you won't have to retype long blocks of code. That includes the following six-line code block:

```
f(x) = x^3 - 9*x^2 + 26*x - 24
g(x) = x^3 - 13*x^2 + 54*x - 72
p1(x) = x^3 - 14*x^2 + 59*x - 70
p2(x) = x^3 - 13*x^2 + 47*x - 35
p3(x) = x^3 - 18*x^2 + 107*x - 210
p4(x) = x^3 - 15*x^2 + 68*x - 84
```

First, use the `factor` command to find the roots of each polynomial. Second, compute the `gcd` of each polynomial with $f(x)$, and of each polynomial with $g(x)$. We say that two polynomials have a "non-trivial gcd" if and only if their gcd is a polynomial of degree one or higher, but we say that two polynomials have a "trivial gcd" if and only if their gcd is a constant. Based on these gcd's and factorizations, figure out the property that will make the following sentence true:

> A polynomial $a(x)$ and a polynomial $b(x)$ will have a trivial gcd if and only if their roots are _____.

1.8. Using Sage to Solve Problems Symbolically

When we say that a computer has solved a problem for us, that can come out to be in one of two flavors: symbolically or numerically. When we solve numerically, we get a decimal expansion for a (very good) approximation of the answer. When we solve symbolically, we get an exact answer, often in terms of radicals or other complicated functions. It is a matter of saying if

the solutions to

$$\frac{x^2}{2} - x - 2 = 0$$

are $1 + \sqrt{5}$ and $1 - \sqrt{5}$ or saying that they are

$$-1.23606797749979 \quad \text{and} \quad 3.23606797749979.$$

The former is an example of a symbolic solution, and the latter a numerical solution. Often, a numerical answer is desired. However, if the answer is a formula, then a symbolic solution is the only way to go. This section is about symbolic solutions, and then the next section ("Using Sage as a Numerical Solver," Section 1.9, starting on page 62) is about numerical solutions.

1.8.1. Solving Single-Variable Formulas

First, let's try solving some single-variable problems. We can type

```
solve(x^2 + 3*x + 2, x)
```

and then we obtain

```
[x == -2, x == -1]
```

It is important to remember the asterisk between the 3 and the `x`. To be precise, to type

```
solve(x^2 + 3x+2, x)
```

would be wrong because of the absence of the asterisk between the "3" and the "`x`." Another, more illustrative, example of a symbolic computation is

```
solve(x^2 + 9*x + 15 == 0, x)
```

which outputs

```
[x == -1/2*sqrt(21) - 9/2, x == 1/2*sqrt(21) - 9/2]
```

By the way, do you notice how the solutions are in a comma-separated list, enclosed in square brackets? That's an example of a list, in notation that Sage inherited from the computer language Python.

One way to really see what symbolic versus exact is about is to compare

```
3+1/(7+1/(15+1/(1+1/(292+1/(1+1/(1+1/6))))))
```

which returns `1,354,394 / 431,117` to

```
N(  3+1/(7+1/(15+1/(1+1/(292+1/(1+1/(1+1/6))))))  )
```

which returns `3.14159265350241`. That could easily be mistaken for π. In fact it is really close to π with relative error 2.78×10^{-11}. (That formula was found using continued fractions, a technique that we will discuss in Section 4.20, starting on page 264.)

There is no restriction to polynomials. You can also type

```
var('theta')
solve(sin(theta)==1/2, theta)
```

to get

```
[theta == 1/6*pi]
```

but note that there are many values for which $\sin\theta = 1/2$. For example,

$$\theta \in \left\{\ldots, -\frac{31\pi}{6}, -\frac{23\pi}{6}, -\frac{19\pi}{6}, -\frac{11\pi}{6}, -\frac{7\pi}{6}, \frac{\pi}{6}, \frac{5\pi}{6}, \frac{13\pi}{6}, \frac{17\pi}{6}, \frac{25\pi}{6}, \frac{29\pi}{6}, \ldots\right\}$$

are several values of θ that solve $\sin\theta = 1/2$.

1.8.2. Declaring Variables

You might be confused to see that `var('theta')` command above. The idea is that you need to tell Sage that this variable is not yet known. We've used variables before, but that's because we had assigned values to them. When you want a variable to represent some unknown quantity, you must use `var`. The variable x is always pre-declared; you do not need to declare it—Sage assumes that x is an unknown, unless we previously assigned it a value.

Sometimes it is convenient to declare several variables at once. We can do this in the following way:

```
var('a b c')
```

That's merely an abbreviation for

```
var('a')
var('b')
var('c')
```

Also note that the following four forms are equivalent:
`var("a, b, c"), var("a b c"), var('a b c'), var('a, b, c')`
and you can use whichever one you prefer.

1.8.3. Labelling Variables with Comments

It is good programming style to notify the reader of your code about the meaning of each variable that you declare. This is an example of "commenting your code," and that's particularly important for long and complicated programs. In Python and Sage, we use the symbol `#` for that, which is sometimes called "the number sign" or "the hash mark." On keyboards in the USA, it is located with the numeral `3`. Here is an example:

```
var('a') # the quadratic coefficient
var('b') # the linear coefficient
var('c') # the constant coefficient
```

You can write anything you want after the `#` symbol, because in both Sage and Python, anything after that symbol until the end of that line is ignored by the computer, because it is intended to be read only by humans.

While not everyone has this habit, it is a good idea in a large program to describe each variable with a comment, unless the names of the variables are self-explanatory. For example, if you name a variable `v`, then you might want to tell someone who is reading your code that `v` stands for velocity. However, if you name the variable `velocity` from the beginning, then a comment is not required.

1.8.4. Solving Multivariable Formulas

Now we're going to use Sage to rederive the quadratic formula. First we must declare our variables with

```
var('a b c')
```

and then we type

```
solve( a*x^2 + b*x + c == 0, x )
```

and receive back

```
[x == -1/2*(b + sqrt(b^2 - 4*a*c))/a,
x == -1/2*(b - sqrt(b^2 - 4*a*c))/a]
```

which has some terms in an unusual order but is undoubtedly correct. Once again, the square brackets and comma indicate a list in Sage.

Of course, you probably know the quadratic formula very well. (At least I hope you do!) However, you probably do not know Cardano's cubic formula. On page 61 in Subsection 1.8.8, we'll see how Sage can "rediscover" Cardano's formula on the spot when similarly programmed.

1.8.5. Linear Systems of Equations

You can solve several equations simultaneously, whether they are linear or non-linear. An easy case might be to solve

$$\begin{aligned} x + b &= 6, \\ x - b &= 4, \end{aligned}$$

which would be done by typing

```
var('b')
solve( [x+b == 6, x-b == 4], x, b )
```

Again, note the use of square brackets and commas to form a list in Sage. You can enclose any data with `[` and `]` and separate the entries with commas to make a list.

Also, it is important to point out that there is no harm in "declaring" b twice. Two `var` commands do no harm to each other. On the other hand, there is also no need to "declare" b twice, as once it is declared, Sage will remember that it is a variable.

A more typical linear system might be

$$\begin{aligned} 9a + 3b + 1c &= 32, \\ 4a + 2b + 1c &= 15, \\ 1a + 1b + 1c &= 6, \end{aligned}$$

and to solve that we'd type

```
var('a, b, c')
solve( [9*a + 3*b + c == 32,   4*a + 2*b + c == 15,
a + b + c == 6], a, b, c )
```

and Sage gives the answer

```
[[a == 4, b == -3, c == 5]]
```

which is correct. Of course, this is exactly the linear system of equations that you would use if someone asked you to find the parabola connecting the points $(3, 32)$ and $(2, 15)$ as well as $(1, 6)$. That would be $f(x) = 4x^2 - 3x + 5$. This is another example of a Vandermonde Matrix (except that we didn't use a matrix here), as we saw in Subsection 1.5.6, starting on page 41.

Naturally, linear systems of equations can also be solved with matrices. In fact, Sage is extremely useful when working with matrices, because it can remove much of the tedium normally associated with matrix algebra. Matrices were discussed in Section 1.5, starting on page 33, but the discussion will continue in Section 4.4, starting on page 201, Section 4.16, starting on page 241, and Subsection 4.24.2, starting on page 289.

1.8.6. Non-Linear Systems of Equations

While linear equations are very easy to solve via matrices, the non-linear case is usually much harder. First, we will warm up with just one equation, but a highly non-linear one. Let us try to solve

$$x^6 - 21x^5 + 175x^4 - 735x^3 + 1{,}624x^2 - 1{,}764x + 720 = 0,$$

which can be done by

```
solve( x^6 - 21*x^5 + 175*x^4 - 735*x^3 + 1_624*x^2 - 1_764*x
      + 720 == 0, x)
```

which gives

```
[x == 5, x == 6, x == 4, x == 2, x == 3, x == 1]
```

That is a list of six answers because that degree 6 polynomial has six roots. It was easy to read this time, but you can also access each answer one at a time. Instead, we type

```
answer = solve( x^6 - 21*x^5 + 175*x^4 - 735*x^3 + 1_624*x^2
               - 1_764*x + 720 == 0, x)
```

and then we can type `print(answer[0])` or `print(answer[1])` to get the first or second entries. To get the fifth entry of that list, we'd type `print(answer[4])`, and similarly, we'd type `print(answer[2])` to get the third entry. This is because Sage is built out of Python, and Python numbers its lists from 0 and not from 1. Most computer languages across the decades have made an identical design decision. It has to do with how the computer allocates memory locations for the several entries of a list. This is one of the many fascinating details of how computers work inside, but it would lead us astray to go into further details on this point. If this sort of discussion interests you, then you should consider taking a course about the internals of computer hardware. Such courses are usually called *Computer Organization* or *Computer Architecture* at most universities in the USA.

Now consider this problem, suggested by Professor Jason Grout, formerly of Drake University. To solve

$$\begin{aligned} p+q &= 9, \\ qy+px &= -6, \\ qy^2+px^2 &= 24, \\ p &= 1, \end{aligned}$$

we would type

```
var('p q y')
eq1 = p+q == 9
eq2 = q*y + p*x == -6
eq3 = q*y^2 + p*x^2 == 24
eq4 = p == 1
solve( [eq1, eq2, eq3, eq4 ], p, q, x, y )
```

which produces

```
[[p== 1, q== 8, x == -4/3*sqrt(10) - 2/3, y ==
1/6*sqrt(10) -2/3], [p== 1, q== 8, x == 4/3*sqrt(10)
- 2/3, y == -1/6*sqrt(10) - 2/3]]
```

As you can see, that is a list of lists, again using square brackets and commas. Clearly, that is a very hard to read mess. Using the technique that we learned when analyzing the degree 6 polynomial above, we replace the last line with

```
answer=solve( [eq1, eq2, eq3, eq4 ], p, q, x, y )
```

Now we can type

```
print( answer[0] )
print( answer[1] )
```

and we get

```
[p==1, q==8, x== -4/3*sqrt(10) - 2/3, y== 1/6*sqrt(10) - 2/3]
[p==1, q==8, x== 4/3*sqrt(10) - 2/3, y== -1/6*sqrt(10) - 2/3]
```

This is Sage's way of telling you

Solution #1: $p = 1, q = 8, x = -\frac{4}{3}\sqrt{10} - \frac{2}{3}, y = \frac{\sqrt{10}}{6} - \frac{2}{3}$,

Solution #2: $p = 1, q = 8, x = \frac{4}{3}\sqrt{10} - \frac{2}{3}, y = -\frac{\sqrt{10}}{6} - \frac{2}{3}$.

Since there are only two answers, we cannot ask for a third answer. In fact, if we type `print( answer[2] )`, then we get

```
Traceback (click to the left of this block for traceback)
...
IndexError: list index out of range
```

which is Sage's way of telling you that it has already given you all the answers for that problem—the list does not contain a third element.

Now let's try to intersect the hyperbola $x^2 - y^2 = 1$ with the ellipse $x^2/4 + y^2/3 = 1$. We type

```
var('y')
solve([x^2-y^2==1,  (x^2)/4+(y^2)/3==1], x, y)
```

and obtain

```
[[x == -4/7*sqrt(7), y == -3/7*sqrt(7)], [x == -4/7*sqrt(7),
y == 3/7*sqrt(7)], [x == 4/7*sqrt(7), y == -3/7*sqrt(7)],
[x == 4/7*sqrt(7), y == 3/7*sqrt(7)]]
```

which is unreadable. Once again, we change the code to be

```
var('y')
answer=solve([x^2-y^2==1,  (x^2)/4+(y^2)/3==1], x, y)
print(answer[0])
print(answer[1])
print(answer[2])
print(answer[3])
print(answer[4])
```

and that produces four answers and an error message

```
[x == -4/7*sqrt(7), y == -3/7*sqrt(7)]
[x == -4/7*sqrt(7), y == 3/7*sqrt(7)]
[x == 4/7*sqrt(7), y == -3/7*sqrt(7)]
[x == 4/7*sqrt(7), y == 3/7*sqrt(7)]
Traceback (click to the left of this block for traceback)
...
IndexError: list index out of range
```

which is Sage's way of telling you

Solution #1: $x = -\frac{4}{7}\sqrt{7}, y = -\frac{3}{7}\sqrt{7}$,

Solution #2: $x = -\frac{4}{7}\sqrt{7}, y = \frac{3}{7}\sqrt{7}$,

Solution #3: $x = \frac{4}{7}\sqrt{7}, y = -\frac{3}{7}\sqrt{7}$,

Solution #4: $x = \frac{4}{7}\sqrt{7}, y = \frac{3}{7}\sqrt{7}$

but that there is no "Solution #5." We will learn about something called "`for` loops," in Subsection 5.1.1, starting on page 319, and that will allow us to write compact but readable code that looks less repetitive than the five nearly identical `print` statements of our previous example. Those of my readers who have programmed before know that `for` loops are both easy to learn and extremely useful. Since we'll cover that later, we do not need to discuss that now.

Because finding the roots of a polynomial (or finding the solutions to an equation) is a common mathematical task, there are some other ways to display the solutions. For example, there's some code which will display each root on its own line, and you can choose to show only real roots, only rational roots, or only integer roots. I will describe that in detail on Subsection 5.9.7, starting on page 407.

Furthermore, we aren't limited to polynomials. We can type

```
solve( log( x^2 ) == 5/3, x )
```

and we receive back

```
[x == -e^(5/6), x == e^(5/6)]
```

both of which satisfy the given equation. Note that `log` in Sage means the natural logarithm, as was explained on page 12 in Subsection 1.2.6.

We can also do

```
solve(sin(x+y)==0.5,x)
```

and obtain

```
[x == 1/6*pi - y]
```

but that's clearly not comprehensive. For example, if $x = 5\pi/6 - y$, then we have

$$\begin{aligned} \sin(x+y) &= \sin\left(\frac{5\pi}{6} - y + y\right) \\ &= \sin\left(\frac{5\pi}{6}\right) \\ &= 1/2. \end{aligned}$$

Now to practice all that we've learned about declaring variables, commenting variables, symbolic solving, and displaying each solution separately, we can update our previous code (from page 56 in Subsection 1.8.4) that derived the quadratic formula for us. Try the following code and see what it does:

```
var('a') # the quadratic coefficient
var('b') # the linear coefficient
var('c') # the constant coefficient
answer = solve( a*x^2 + b*x + c, x )
show(answer[0])
show(answer[1])
```

1.8.7. Polynomials and the Complex Numbers

Normally, we expect an eighth-degree polynomial to have 8 roots, over the complex plane, unless some are repeated roots. Therefore, if we ask, "What are the roots of $x^8 + 1$?" then surely we expect 8 roots. These are the 8 eighth-roots of -1 in the set of complex numbers. We can find them with

```
solve(x^8 == -1, x)
```

which produces

```
[x == (1/2*I + 1/2)*(-1)^(1/8)*sqrt(2), x == I*(-1)^(1/8), x ==
(1/2*I - 1/2)*(-1)^(1/8)*sqrt(2), x == -(-1)^(1/8), x == -(1/2*I +
1/2)*(-1)^(1/8)*sqrt(2), x == -I*(-1)^(1/8), x == -(1/2*I -
1/2)*(-1)^(1/8)*sqrt(2), x == (-1)^(1/8)]
```

giving us eight distinct complex numbers, all of which have -1 as their eighth power. This can be calculated (by hand) more slowly with DeMoivre's formula, but for Sage, this is an easy problem. I simply copy-and-paste the first root and raise it to the eighth power, as

```
(  (1/2*I + 1/2)*(-1)^(1/8)*sqrt(2)  )^8
```

and Sage returns the answer -1. I can check the other seven roots by cutting-and-pasting similarly.

1.8.8. Cardano's Formula and the Quartic Formula

To see why we don't ask undergraduates to memorize Cardano's cubic formula, try typing in

```
var('b c')
solve(x^3 + b*x + c==0, x)
```

That gives you Cardano's formula for a monic depressed cubic. A cubic polynomial is said to be depressed if the quadratic coefficient is zero, and monic means that the leading coefficient (here, the cubic coefficient) is one.

The command `show` can be used to make equations much more readable. For example, we can see the above formula more clearly with the following code:

```
var('b c')
answer = solve( x^3 + b*x + c, x )
show(answer[0])
show(answer[1])
show(answer[2])
```

The full version of the cubic formula, which does not require the leading coefficient to be one, or the second-to-leading coefficient to be zero, would be given by

```
var('a b c d')
solve(a*x^3 + b*x^2 + c*x + d == 0, x)
```

which is just insanely complicated. Try it, and you'll see. In fact, it is so complicated that one would have to confess that it is not useable.

An even uglier formula would be the general formula for a quartic, or degree 4, polynomial. Surely any quartic can be written

$$a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0 = 0$$

so we can ask Sage to solve it with

```
var('a0 a1 a2 a3 a4')
solve( a4*x^4 + a3*x^3 + a2*x^2 + a1*x + a0 == 0, x)
```

and we get an answer that is a formula of horrific proportions.

However, Sage can use that (enormous) formula to solve particular quartic equations exactly. Try the following code, and see what happens:

```
answer = solve( x^4 - 20*x^3 + 3*x^2 + x + 5, x )
show(answer[0])
show(answer[1])
show(answer[2])
show(answer[3])
```

1.8.9. A Challenge: The Reversals of Polynomials

We're going to consider another property of polynomials now. Let's define the reversal of a polynomial by the following operation:

$$16x^4-10x^3-463x^2+1018x-120 \text{ becomes } -120x^4+1018x^3-463x^2-10x+16$$

As you can see, the highest-degree coefficient traded places with the lowest-degree coefficient, the second-highest-degree coefficient traded places with the second-lowest-degree coefficient, and so forth. To aid in cutting and pasting, here is that polynomial again, but in Sage notation, along with two others that you can experiment with. I've even typed the reversals for you.

```
p1(x) = 16*x^4 - 10*x^3 - 463*x^2 + 1_018*x - 120
p2(x) = 30*x^5 - 517*x^4 + 69*x^3 + 849*x^2 + 29*x - 204
p3(x) = -80*x^5 - 974*x^4 - 927*x^3 + 1_265*x^2 + 947*x - 231
r1(x) = -120*x^4 + 1_018*x^3 - 463*x^2 - 10*x + 16
r2(x) = -204*x^5 + 29*x^4 + 849*x^3 + 69*x^2 - 517*x + 30
r3(x) = -231*x^5 + 947*x^4 + 1_265*x^3 - 927*x^2 - 974*x - 80
```

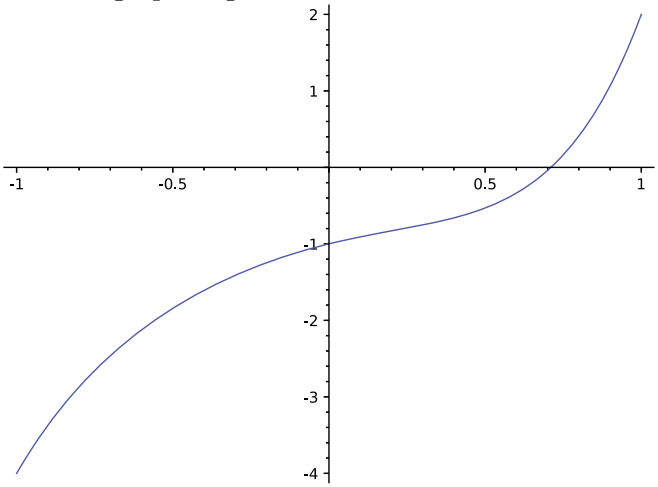
For each polynomial, use the `solve` command to symbolically find all its roots. Then use it again to find the roots of its reversal. From this data, you should be able to deduce the general rule for how the roots of a polynomial are related to the roots of its reversal. You can ask any mathematics professor for the proof of this theorem. Technically, we must assume that the constant coefficient does not equal zero; otherwise the reversal is not easy to define.

1.9. Using Sage as a Numerical Solver

The set of x-values that make $f(x) = 0$ are commonly called "the roots" of $f(x)$. Let's suppose you need to know the value of a root of

$$f(x) = x^5 + x^4 + x^3 - x^2 + x - 1,$$

a polynomial whose graph is given as follows:



Furthermore, suppose you are interested in a root between -1 and 1. This could come about because you graphed it (with or without Sage) and saw that there is a root in that region. Or it could come about because you saw that $f(-1) = -4$ but $f(1) = 2$—thus f clearly crosses the x-axis at some point between -1 and 1, though we don't know where just yet. Perhaps a problem in a textbook simply tells you that the root is between -1 and 1.

To solve for that root, you need only type

```
find_root(x^5 + x^4 + x^3 - x^2 + x - 1, -1, 1)
```

and Sage tells you that

$$0.71043425578721398$$

is the root. This is a numerical approximation because quintic polynomials have a very special property—if you don't know what the special property is, we will discuss it in Subsection 1.9.4, starting on page 68.

While it is an approximation because the computer does several billion instructions per second, Sage invests computation time in refining the approximation. Therefore it is an approximation that is trustworthy to around an accuracy of 10^{-16}, which is one-tenth of a quadrillionth. This is a good approximation, by any standard.

Meanwhile, it turns out there's no root between -1 and 0. Perhaps you are told that, or perhaps you graph it (with or without Sage). If you type

```
find_root(x^5 + x^4 + x^3 - x^2 + x - 1, -1, 0)
```

then Sage tells you

```
RuntimeError: f appears to have no zero on the interval
```

which is perfectly honest because there's no root in the interval for Sage to

go and find! If you have no idea where the root is, you can type

```
find_root(x^5 + x^4 + x^3 - x^2 + x - 1, -10^12, 10^12)
```

which is asking Sage to find a root that is between plus/minus one trillion.

Using the `find_root` command, extremely sophisticated problems can be solved. Here's a favorite: If someone asks you about $x^x = 5$, then you could try to find where $x^x - 5 = 0$. You would type

```
find_root(x^x - 5,  1,  10)
```

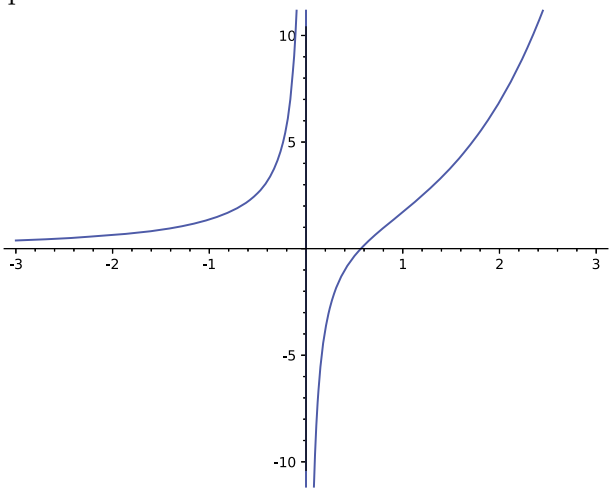
or equivalently

```
find_root(x^x == 5,  1,  10)
```

Another one is to find where $e^x = 1/x$ and you'd do that by finding a root of $e^x - 1/x = 0$. You could graph that with

```
plot( e^x - 1/x, -3, 3, ymin=-10, ymax=10)
```

obtaining the plot



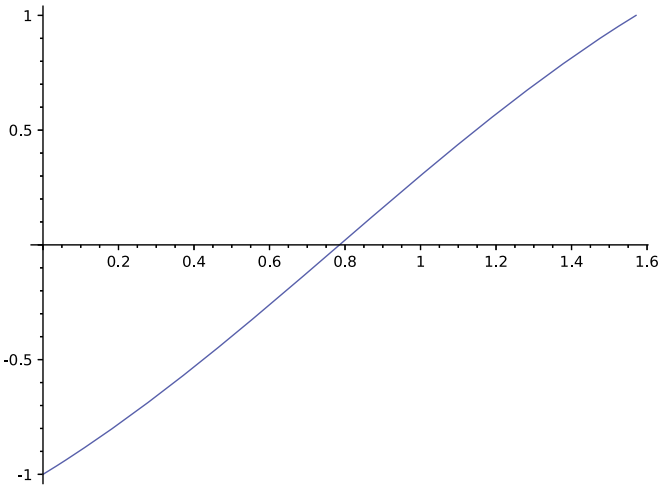
and then you can see the root is between -1 and 1. This means that typing

```
find_root(e^x - 1/x, -1, 1)
```

will result in finding out that the root is at $x = 0.56714329040979539$.

Last but not least, another favorite is to find where $\sin x = \cos x$, or equivalently where $(\sin x) - (\cos x) = 0$. It happens to be the case that there's such an x between 0 and $\pi/2$, which you can see by graphing. The

graph is



Now here you could ask for a root of $f(x) = (\sin x) - (\cos x)$. Alternatively, you can also type

```
find_root( sin(x) == cos(x), 0, pi/2 )
```

which is just easier notation. The answer is $\pi/4$, though it is hard to recognize here, since it is given as a decimal.

1.9.1. Returning to an Old Problem about Compound Interest

Earlier, on page 13 in Subsection 1.2.7, we saw a problem involving compound interest and finding how many months will be required to turn \$5,000 into \$7,000. Now let's see how to solve that with Sage in only one line, rather than the previous approach which required the human to do some algebra.

The first line of our analysis was

$$7{,}000 = 5{,}000(1 + 0.045/12)^n$$

and translating that into the language of Sage results in

```
7_000 == 5_000*(1+0.045/12)^n
```

Then we should type

```
var('n')
find_root( 7_000 == 5_000*(1+0.045/12)^n, 0, 10_000)
```

Note that the second line means that I'm searching for a value of n that makes the equation satisfied and that the value of n should be between 0 and 10,000. This seems reasonable because 10,000 months is a little over 833 years—surely it won't take that long. Note that it is vital to not include a comma (or period) inside of 10,000, as Sage's syntax forbids commas and periods from serving as thousands-separators, as we saw on page 2 in Subsection 1.1.2 and page 13 in Subsection 1.2.8. The first command above, `var('n')`, is just declaring n as a variable, which we saw on page 55 in

Subsection 1.8.2. In any case, we get the answer `89.894060933080027` as before and learn that 90 months will be required.

1.9.2. Solving for Multiple Roots

As we have seen, each use of the `find_root` command will find only one root, but of course even simple functions like polynomials can have multiple roots. Suppose we were asked to find all the roots of

$$f(x) = x^4 - 8x^3 + 23x^2 - 28x + 12$$

and we have an initial hint that all the roots are inside the interval $-10 < x < 10$.

We should type the code

```
f(x) = x^4 - 8*x^3 + 23*x^2 - 28*x + 12
find_root( f(x)==0, -10, 10 )
```

and we learn that a root is $x = 3.0$. Now we can replace the `10` with `3`, and we get $x = 3.0$ as an answer yet again. Therefore, let's replace `3` with `3-0.0001` as our upper bound. This results in

```
0.9999999999999881
```

We might try replacing the bounds with `-10, 0.9999` but then we receive a long and incomprehensible error message, ending with

```
RuntimeError: f appears to have no zero on the interval
```

The astute reader will realize that we did not check the interval between the two roots that we had found. So we should use `1, 3-0.0001`, and we obtain $x = 1.0$ again. Therefore, we try `1+0.0001, 3-0.0001` and we obtain

```
2.0000000000177662
```

It turns out that we will not obtain any additional roots by searching further. This is a bit of a surprise, because $f(x)$ is a fourth-degree polynomial. Therefore, unless there is a repeated root, we expect one of three cases:

- four real roots and no complex/imaginary roots,
- two real roots and two complex/imaginary roots,
- no real roots and four complex/imaginary roots.

So then we are forced to wonder if perhaps there is a fourth root somewhere out there. We can use the symbolic solver that we learned in Section 1.8, starting on page 53, as follows:

```
f(x) = x^4 - 8*x^3 + 23*x^2 - 28*x + 12
solve( f(x)==0, x )
```

This results in the output

```
[x == 3, x == 1, x == 2]
```

confirming our belief that there are only three roots. Finally, we can get at the heart of the matter with

```
f(x) = x^4 - 8*x^3 + 23*x^2 - 28*x + 12
factor(f(x))
```

and we receive the output

```
(x - 1)*(x - 2)^2*(x - 3)
```

We now understand that we have a root at $x = 2$ of multiplicity two, as well as two roots of multiplicity one at $x = 1$ and $x = 3$. Since $2 + 1 + 1 = 4$, this is a valid possibility for a fourth-degree polynomial.

1.9.3. A Challenge: Human-Assisted Root Finding

Using the `find_root` command, find the three smallest positive real numbers such that $\sec x = e^x$. Yet beware of the fact that you will need to help `find_root` find those solutions, by narrowing down the search interval. For example, if you ask `find_root` to search in the interval $5 < x < 10$, then it will report

```
RuntimeError: f appears to have no zero on the interval
```

even though one of the solutions is indeed inside the interval $5 < x < 10$.

Be certain to check your work, such as with the following code:

```
print( sec(2.3456789) )
print( e^(2.3456789) )
```

Because 2.3456789 is not a number where $e^x \approx \sec x$, you'll see two outputs that are relatively far apart. However, if you find a solution and replace 2.3456789 with that solution, then you will see two outputs that are very close together. Specifically, using only Sage commands that we've learned so far, I was able to get the first solution to have e^x and $\sec x$ match to 13 significant digits, the second solution to have them match to 11 significant digits, and the third solution to have them match to 9 significant digits. Remember, we are looking for positive real solutions, so $x = 0$ is not included, even though it is obviously a solution to $\sec x = e^x$.

Unfortunately, Sage was not able to locate the fourth smallest solution at all. You might be wondering how I know that a fourth positive real solution exists. The code

```
print( sec(20.4203512483327) )
print( exp(20.4203512483327) )
print( 'compared with' )
print( sec(20.4203522483327) )
print( exp(20.4203522483327) )
```

produces the output

```
999999.044273481
7.38662183837371e8
compared with
1.04744985205174e12
7.38662922499925e8
```

Since $\sec x$ was smaller than e^x for 20.4203512483327 and since $\sec x$ was larger than e^x for 20.4203522483327, we know that there must be a real number x such that $\sec x = e^x$ inside the interval

$$20.4203512483327 < x < 20.4203522483327$$

yet, Sage cannot find it, using `find_root` as implemented when I wrote this challenge on June 25, 2020. Of course, it is possible that improvements to `find_root` could be added in later versions of Sage.

Now I have revealed to you that the first, second, and third solutions must be between 0 and 20.4203512483327. Challenge yourself to go and find those solutions.

1.9.4. A Theoretical Interlude, Part One

This subsection is absolutely worth reading, but depending on your mathematical background, you might find it rather hard to understand. Rest assured that you can understand essentially everything else in this book without understanding anything from this subsection at all. Many readers will want to skip this entirely and proceed with Section 1.10, starting on page 70.

If someone attempts a PhD in mathematics, then among many other topics of coursework, they will take 1–3 years of coursework (spread across both the undergraduate program and the graduate program) in a subject usually called *Abstract Algebra*, sometimes called *Modern Algebra*, but more usefully called the *Theory of Groups, Rings, and Fields*. Galois[14] theory, an important topic of abstract algebra uniting group theory and field theory, has a theorem called the Abel–Ruffini Theorem.

Named for Paolo Ruffini (1765–1822) and Niels Henrik Abel (1802–1829), this theorem guarantees that there will never exist a general formula for computing the roots of all fifth-degree polynomials exactly (where the formula is not infinitely long and is composed of additions, subtractions, multiplications, divisions, integer exponents, and the extraction of square roots, cube roots, and higher roots). The restriction that I have given in parentheses is important, because using elliptic functions, a general formula for the

[14]Galois theory is named for Évariste Galois (1811–1832). The reader has probably noticed that Galois and Abel both died very young (at ages 20 and 26, respectively) but they nonetheless made epic permanent contributions to the development of mathematics. One cannot help but wonder what they could have discovered if they hadn't died so young.

exact computation of the roots of all fifth-degree polynomials does exist—a fact proven in 1858 by Charles Hermite (1822–1901). The word "exactly" is important too, because Newton's method and other techniques can produce numerical approximations. The impossibility result of the Abel–Ruffini theorem also extends to all higher degrees, so there is nothing particularly defective about the fifth degree. By the way, the technical term for a fifth-degree polynomial is a *quintic polynomial*, just as the technical term for a second-degree polynomial is a quadratic polynomial.

Moreover, it should be noted that these "impossible" quintic polynomials are not hideously complicated. We have the example

$$x^5 - x + 1$$

whose roots cannot be found exactly by any formula (where the formula is not infinitely long and is composed of additions, subtractions, multiplications, divisions, integer exponents, and the extraction of square roots, cube roots, and higher roots).

The famous "rational roots theorem" can easily produce all rational roots of all polynomials with rational coefficients, regardless of degree, and it is relatively easy to learn. So if even one root of a quintic polynomial is rational, then the rational roots theorem can be used to find that root. Polynomial long division can then give a quartic polynomial (a fourth-degree polynomial) whose roots are the other four roots of the original quintic. Since techniques for finding the roots of quartic polynomials are known (and already programmed into Sage), then all the roots of the original quintic can be found. Therefore, we have now proven that any "impossible" quintic polynomial has only irrational roots, and no rational roots.

I find it intriguing that there are non-impossible quintic polynomials whose every root is irrational. Therefore, the condition of all the roots being irrational is necessary, but not sufficient, for a quintic to be impossible. If you type the code

```
solve( (x^2 - 3)*(x^3 - 2), x )
```

into Sage, then you'll see that all five roots have been instantaneously found and written exactly, even though all five are irrational. By the way, the term "impossible quintic polynomial" is casual. The formal term is "unsolvable quintic polynomial."

While the Abel–Ruffini Theorem is highly regarded as the crown jewel of Galois theory, there are many other uses for that theory, including some other proofs of impossibility related to geometry topics. For our purposes in Sage, it is sufficient to remember that symbolic methods for computing roots of polynomials exactly cannot be expected to work in all cases, when polynomials of degree five or higher are involved.

The Abel–Ruffini Theorem, more often called "the unsolvability of quintic polynomials," is in stark contrast to quadratic polynomials, whose solution was known to the Babylonians (if not earlier) and to cubic polynomials

(third degree), whose solution was known in Renaissance Europe. Today it is called Cardano's cubic formula, which we caused Sage to derive on page 61 in Subsection 1.8.8.

Cardano's cubic formula is named for Gerolamo Cardano (1501–1576), but that is somewhat unfair. A sequence of mathematicians had worked on cubic equations over the centuries—so many that it would be difficult to summarize in a concise way. Suffice it to say that Cardano merely published techniques that had been developed by others, including one mathematician, Niccolò Tartaglia (1500–1557), who had sworn Cardano to secrecy before revealing his methods. A nice 3-page article on this story is "The Cardano-Tartaglia dispute" by Richard Feldmann, published in *The Mathematics Teacher* in March 1961 [**21**].

There are aftershocks of the Abel–Ruffini Theorem throughout mathematics. Some of my readers will not understand the rest of this paragraph, but if you know what an eigenvalue of a matrix is (to be discussed in Subsection 4.16.1, starting on page 241), then you might be intrigued by the following result: There will never exist a general formula for exactly computing any of the eigenvalues of 5×5 matrices with integer entries (where the formula is not infinitely long and is composed of additions, subtractions, multiplications, divisions, integer exponents, and the extraction of square roots, cube roots, and higher roots). The reason is that if there were such a formula, then that would easily give rise to a method for finding exactly the roots of arbitrary quintic polynomials with integer coefficients. Yet, the Abel–Ruffini Theorem guarantees that this is impossible.

Vaguely speaking, proving that some task is impossible to accomplish is among the hardest categories of proof to write. This discussion will continue in Subsection 1.12.10, starting on page 84.

1.10. Getting Help When You Need It

By now we have learned more than a few commands and it is important to discuss how you can use the built-in "help features" for Sage. There are several, each intended for a different purpose.

1.10.1. The Question Mark (?) Operator

Now let's imagine that I have remembered the name of the command—perhaps `solve`—but not the exact syntax. Then what I should type is `solve?` but with no space between the question mark (`?`) and the command, then click "Evaluate" (or press Shift-Enter). Sage will display lots of great information about the command. Feel free to read the discussion if you like, but I find it faster to merely scroll toward the examples and see how things are done there, mimicking what I find there as needed. By the way, the huge collection of information found in that window is called "the docstring," which is short for "the documentation string."

Most Sage users, including myself and even experts, will frequently refer to the docstring, because it is simply impossible to remember the exact syntax of the enormous number of commands available. You might also like to keep this book nearby when you are using Sage, to look up the syntax whenever you might need it.

1.10.2. Displaying the Source Code of a Command

There is another way to use the `?` operator. If you put two question marks after a command name, such as `N??`, then you will see the entire source code (usually written in the computer language Python) for the command. In this case you'll see the Python code for the `N()` command which turns exact algebraic expressions into floating point numbers. Of course, the code is very advanced and would require much more knowledge of Python than this book will attempt to impart. (We will learn basic-level and intermediate-level Python programming in Chapter 5, starting on page 317.)

Nonetheless, it is important that this feature (of displaying source code) is there for philosophical reasons. Often we have to know *how* a mathematical program will accomplish its task (in other words, how it will work internally) in order to figure out how to best format and preprocess our data prior to using that program. This comes up in all sorts of situations, and having access to the source code is essential. It also allows the ideas in the mathematical program to be used by other programmers in their own programs. In this way, the open-source community lives as a sort of "hive mind" sharing ideas widely. Also, mathematical software will inevitably have bugs. A colleague of mine found a bug in a well-known commercial computational software product and notified the company via email about it. They acknowledged the bug, but three years later, it remained unfixed. In Sage, you can simply fix the bug yourself and submit the change for peer review. If the change is found to be helpful and valid, it will be made. Sometimes the entire process can take as little as six weeks.

This now brings up an important point: To use either of those first two methods of getting help, you have to know (exactly) the name of the command. What if you cannot remember the name of the command at all? This happens often, and there are six common solutions.

- You can check in this book, using the table of contents or the index.
- You can use something called "tab completion," when you remember the first few letters of the command, but not the rest.
- You can search among all the docstrings in Sage for a phrase.
- You can consult the official Sage documentation.
- You can seek out a Sage tutorial focused on that branch of math.
- At `ask.sagemath.org` you can consult the Sage community.

The rest of this section will now explain these options.

1.10.3. Tab Completion

This is a great feature of Sage, but at the moment when these words were typed (July 2020), tab completion works on `CoCalc.com` and local installations of Sage, but not on SageMathCell. If you are typing a long command, like `numerical_approx` for example, and part way through you forget the exact ending (is it `approximation`? `approximations`? `appr`?), then you can hit the tab button. If only one command in the entire library has that prefix, then Sage will fill in the rest for you. If not, then you get a list of suggestions. It is a very useful feature when you cannot remember the exact names of commands.

As another example, perhaps I am confused if the correct command is `find_root`, `findroot`, `find_roots`, `root_find`, or maybe something else entirely. I would type `find` and then press the tab key on my keyboard (without pressing Shift-Enter). I would then get a list of choices, most of which have nothing to do with my intentions. However, one is the command that I seek, `find_root`. Alternatively, if I had reached `find_r` before pressing the tab key, then there is only one possible command, and therefore Sage just completes the command for me.

1.10.4. Searching All the Docstrings for a Mathematical Phrase

There is also a way to search all the docstrings of all the commands in Sage. However, this will usually produce a very large number of results. Type, for example,

```
search_doc('laplace')
```

and you will see a huge response. It is not exactly human-readable. Here is a particular line of that output:

```
en/constructions/calculus.html:228:<div class="section" id="laplace-transforms">
```

The entire line refers to a webpage. The middle number, between the colons, is a line number—one is the top line, two is the second line, and so forth. So in this case, the word "laplace" is found in the 228th line of the `calculus.html` file. The information after the second colon is written in the HTML language. If you know HTML[15], then you can use that information.

So far `search_doc` has not told us anything useful, but the information to the left of the first colon is very useful. That is a filename inside of a directory hierarchy that contains the documentation of Sage. Usually Sage is used through a server, such as SageMathCell or `CoCalc.com`. However, one can do a local installation, which means you are running Sage contained entirely inside your own computer and not communicating with a remote server. This is not recommended except for moderately or extremely experienced users.

[15]If you do not know HTML, perhaps you might want to consider learning it. The HTML language is extremely useful and easy. Being able to make your own webpages is an excellent skill.

What, then, should the rest of us (who use Sage through SageMathCell or `CoCalc.com`) do in order to get at the information? We must go to the following URL, which is essentially the information which `search_doc` provided, but in a different word order:

`https://doc.sagemath.org/html/en/constructions/calculus.html`

As you can see, in order to construct that URL, we just prepended the following prefix to the address returned by `search_doc`

`https://doc.sagemath.org/html/`

That prefix is the address of the Sage documentation (in HTML), and this substitution will work in general. By the way, the `en` refers to the English language, with `fr` referring to French, `de` to German, and `ru` to Russian, according to the normal two-letter language codes in use on the internet, established by `RFC-1766` in 1995.

1.10.5. The Official Sage Documentation

The official documentation of Sage can be found at the address

`https://doc.sagemath.org/`

Many resources are available, including HTML and PDF formats, and are written in various languages. However, the most complete and up-to-date documentation is in English. In particular, the official *Sage Reference Manual* can be found at

`https://doc.sagemath.org/html/en/reference/index.html`

but it is not for beginners! Be warned, the manual is literally thousands of pages long. The manual is like an encyclopedia: You don't read it cover to cover—instead you look up what you need. Luckily, there is a handy "Quick Search" box on the left-hand side.

1.10.6. Tutorials Focused on Sage

Tutorials are much more useful for most readers. Some are organized around a mathematical topic and others are organized around a target audience. There is a collection of Sage tutorials, a partial list of which can be found on the website for this book, under the heading 'Appendix W: Links to Other Websites with Resources for Sage'. At any given moment, some tutorials will probably be out of date for many reasons, but especially because of the significant changes in Sage syntax that occurred during December 2019 and January 2020.

1.10.7. The Community at ask.sagemath.org

There is a wonderful community of active Sage users available to answer questions. You can post a question there, and if you provide sufficient detail, then you'll often find a complete and exhaustive response within 1–3 days. You can also search among previous questions and see their answers. The community can be found at

`https://ask.sagemath.org/`

1.10.8. A Challenge: Exploring a New Command

I'd like you to get some experience with docstrings now. For example, I haven't taught you the command `plot3d` because 3-dimensional plots often look terrible in a black and white printed book, and they often don't look great in a 2-dimensional PDF file either. Nonetheless, pull up the docstring for `plot3d` using the `?` operator. Select one of the examples, and try it out. Enjoy!

Several techniques of 3-dimensional plotting will be covered in the online appendix to this book "Plotting in Color and in 3D" available on this book's website:

`http://www.sage-for-undergraduates.org`

1.11. Using Sage to Take Derivatives

The command for the derivative is just `diff`. For example, to take the derivative of $f(x) = x^3 - x$ you would type

```
diff( x^3 - x, x)
```

to learn the answer

```
3*x^2 - 1
```

Or for $f(x) = \sin(x^2)$ you would type

```
diff( sin(x^2), x )
```

to receive the answer

```
2*x*cos(x^2)
```

Sometimes you might have defined your own function by

```
g(x)=e^(-10*x)
```

earlier, so you can then type

```
diff(g(x), x)
```

to get the derivative, which is naturally

```
-10*e^(-10*x)
```

but you can also type

```
gprime(x) = diff(g(x), x)
```

which is silent, in the sense that it outputs nothing. However, you can type `gprime(2)` or `gprime(3)` and it does what you expect. It tells you the values of $g'(2)$ and $g'(3)$. You can also type

```
g(x)=e^(-10*x)
g(x).derivative()
```

We cannot, however, define a function called `g'(x)`. That's because Sage is built on top of the already existing computer language called Python, and the apostrophe has a predefined meaning in Python. This is unfortunate, but we can always write `gprime(x)` or `g_prime(x)`, so it will not be a barrier for us.

Sage can also do very difficult derivatives:

```
diff( x^x, x)
```

This provides the answer

```
(log(x) + 1)*x^x
```

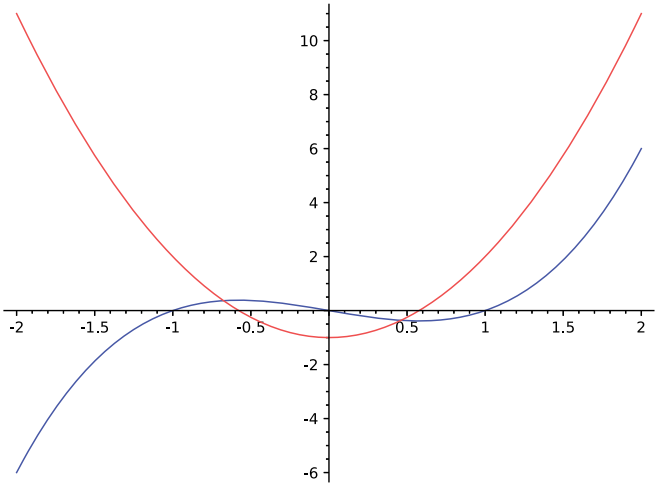
Now if you're interested in giving your calculus skills a workout, but possibly a very intense one, you can try to see how you would calculate the derivative of x^x with your pencil. By the way, it is important to note that `log` refers to the natural logarithm, not the common logarithm, as was explained on page 12 in Subsection 1.2.6.

1.11.1. Plotting $f(x)$ and $f'(x)$ Together

One of my favorite bits of code in Sage is

```
f(x)=x^3-x
fprime(x)=f(x).derivative()
plot( f(x),-2,2, color='blue') + plot( fprime(x),-2,2, color='red')
```

and I use this rather frequently. This plots $f(x)$ and $f'(x)$ on the same graph, with $f(x)$ in blue and $f'(x)$ in red. You can very easily change what function you are analyzing by changing that first line. Here we are analyzing $f(x) = x^3 - x$, in blue (bottom graph), and its derivative, $f(x) = 3x^2 - 1$, in red (top graph). Note, the color is not visible in this black and white text. The plot is as follows:



1.11.2. Higher-Order Derivatives

While you can find the second derivative by using `diff` twice, you can jump directly to the second derivative with

```
diff( x^3-x, x, 2)
```

to get the answer

```
6*x
```

Third or higher derivatives are also no problem. For example, the third derivative can be found with

```
diff( x^3-x, x, 3)
```

For those who don't mind typing more, you can do

```
derivative( x^3-x, x, 2)
```

which is more readable than writing `diff`.

1.11.3. A Challenge: Taking Derivatives

This might seem like it is too easy, but I think it is well worth doing. Consider the function

$$f(x) = e^{2x}\left(x^4 - 2x^3 + 3x^2 - 4x + 5\right)$$

and compute the first, second, third, and fourth derivatives by hand with a pencil and paper. Then have Sage compute those derivatives, defining them as `f_p(x)`, `f_p_p(x)`, `f_p_p_p(x)`, and `f_p_p_p_p(x)`. Of course, there are many different ways to write the same function, depending on simplification. We can use Sage to check if our hand-calculated results are correct. Enter your hand-calculated derivatives as `g_p(x)`, `g_p_p(x)`, `g_p_p_p(x)`, and `g_p_p_p_p(x)`. You can then check your fourth derivative as follows:

```
print( expand( f_p_p_p_p(x) - g_p_p_p_p(x) ) )
```

and similarly for the third, second, and first derivatives.

Don't be surprised if you are wrong on the first try. Keep correcting all your work, fixing your errors, until the expand reveals the answer to be zero. This is exactly the kind of checking of one's work that applied mathematicians working in industry must do, in order to validate their computer models.

1.12. Using Sage to Calculate Integrals

The only thing you need to understand in order to do integrals in Sage is that there are really three types of integrals that can be done: a numerical approximate integral, an exact definite integral, and an indefinite integral. The first two types result in numbers, but they are found approximately or exactly, respectively. The third one is the symbolic antiderivative, which means that the answer is a function—in fact, the answer is a function whose derivative is what you had typed in. If this is confusing, then let us consider a simple example.

1.12.1. An Easy Example of Integration

Using an integral, how would you calculate the area between the x-axis and $f(x) = x^2 + 1$ on the interval $3 \le x \le 6$? Well, that integral would be

$$\int_3^6 \left(x^2+1\right)\, dx = \left(\frac{1}{3}x^3 + x\right)\Big|_3^6 = \left(\frac{1}{3}6^3 + 6\right) - \left(\frac{1}{3}3^3 + 3\right) = 72 + 6 - (9+3) = 66$$

and that is a number. Here we've done it analytically, using calculus. This is an exact definite integral. It is definite because it results in a number. It is exact because we used no approximations.

On the other hand,

$$\int \left(x^2+1\right)\ dx = \frac{1}{3}x^3 + x + C$$

is an indefinite integral. It produces a function, and not a number. Of course, we often calculate an indefinite integral on the way to calculating a definite integral; nonetheless, the definite and indefinite integrals are two distinct categories. If you want a function, then you should use the indefinite integral, and if you want a number, then you should use the definite integral.

1.12.2. The Indefinite Integral

The indefinite integral is the simplest. For example, if you wanted to know

$$\int x\sin(x^2)\ dx$$

you would simply type

```
integral( x*sin(x^2), x )
```

and then you'd learn the answer is

```
-1/2*cos(x^2)
```

Similarly, if you wanted to know

$$\int \frac{x}{x^2+1}dx$$

you would simply type

```
integral( x/(x^2 + 1), x )
```

and then you'd learn the answer is

```
1/2*log(x^2 + 1)
```

1.12.3. More About the $+C$

Let's consider the set of functions whose derivative is $3x^2$. There's an infinite number of them, including x^3+5, x^3-8, x^3+81, $x^3+\sqrt{\pi}$, and so forth. That's why calculus professors insist that you write

$$\int 3x^2\ dx = x^3 + C$$

when computing an indefinite integral.

However, Sage does not know about $+C$. There's actually a fairly good reason for this. It has to do with the fact that it isn't really easy to make a Python function that represents an infinite family of functions. For example, if I define $f(x) = \int 3x^2\ dx$, then what should Python say when I ask for $f(2)$? How can Sage know what member of the family I am referring to?

With this in mind, you have to keep track of the $+C$'s yourself. Correctly handling the $+C$'s is a key task in one of the projects (the project about ballistic projectiles) in Section 2.5, starting on page 120.

1.12.4. The Definite Integral

Alternatively, we could put a lower bound of 0 and an upper bound of 1 on that integral and then get

$$\int_0^1 \frac{x}{x^2+1} dx.$$

To get Sage to compute that definite integral for us, we type

```
integral( x/(x^2 + 1), x, 0, 1 )
```

and we get the answer

```
1/2*log(2)
```

However, remember that "log" means the natural logarithm, not the common logarithm, as explained on page 12 in Subsection 1.2.6.

1.12.5. Integration by Partial Fractions

One thing that is very important in problems that arise from *Differential Equations* and *Calculus II* is the question of integration by partial fractions. If you want to know the partial fraction breakdown of

$$\frac{x^3 - x}{x^2 + 5x + 6},$$

then you must do

```
f(x)=(x^3-x)/(x^2+5*x+6)
f(x).partial_fraction()
```

and get the answer

```
x - 6/(x + 2) + 24/(x + 3) - 5
```

which is correct. In fact, we just calculated

$$\frac{x^3 - x}{x^2 + 5x + 6} = x - \frac{6}{x+2} + \frac{24}{x+3} - 5.$$

However, you can also do the shortcut and ask instead

```
integral( (x^3-x)/(x^2+5*x+6), x )
```

which will result in

```
1/2*x^2 - 5*x - 6*log(x + 2) + 24*log(x + 3)
```

1.12.6. Improper Integrals

One type of improper integral is

$$\int_2^\infty \frac{1}{x^2}\,dx$$

and the way to write that in Sage is to type

```
integral(1/x^2, x, 2, oo)
```

to which Sage responds 1/2.

As you can see, `oo` is a special code for "infinity." It is two letter o's, and the idea is that if you squint, then `oo` looks like an infinity symbol.

Another Convergent Improper Integral:

Another example of an improper integral is

$$\int_{-\infty}^\infty e^{-x^2}dx$$

which in Sage is computed with

```
integral(exp(-x^2),x,-oo,oo)
```

Sage gives the correct response of $\sqrt{\pi}$.

This integral is of great importance in statistics, and we will discuss it again on page 86 in Subsection 1.12.10. Since this improper integral results in a number, we call it "convergent," because the limit converges to that number ($\sqrt{\pi}$, in this case). In contrast, if the limit diverges, then we call the integral "divergent."

A Divergent Improper Integral:

While the integral

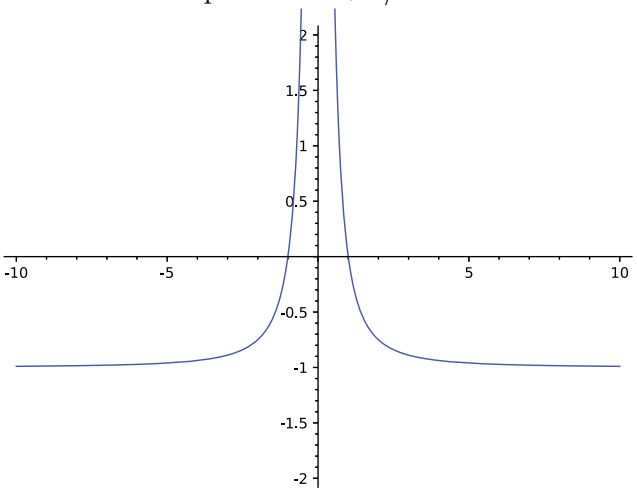
$$\int \left(-1+\frac{1}{x^2}\right)\,dx = -x - \frac{1}{x} + C$$

is mathematically valid and the right-hand side can be evaluated at $x = -1$ and $x = 1$, there is no finite answer to

$$\int_{-1}^1 \left(-1+\frac{1}{x^2}\right)\,dx$$

because this improper integral is divergent. The graph will help reveal why

that is the case. Here is the plot of $-1 + 1/x^2$



generated by

```
plot( -1+1/x^2, -10, 10, ymin=-2, ymax=2 )
```

Yet, Sage knows this and will say

```
-x - 1/x
```

in response to

```
integral(-1+1/x^2,x)
```

but instead will respond

```
ValueError: Integral is divergent
```

to the inquiry

```
integral(-1+1/x^2,x,-1,1)
```

If you'd like a proof of the fact that the above integral is divergent, consider that

$$\int_{-1}^{-1} \left(-1 + \frac{1}{x^2}\right)\, dx = \int_{-1}^{0} \left(-1 + \frac{1}{x^2}\right)\, dx + \int_{0}^{1} \left(-1 + \frac{1}{x^2}\right)\, dx.$$

Let's consider the integral above, from 0 to 1, by means of computing that integral from a to 1, where a will be slightly greater than 0:

$$\begin{aligned} \int_a^1 \left(-1 + \frac{1}{x^2}\right) &= \left(-x - \frac{1}{x}\right)\Big|_a^1 \\ &= \left(-1 - \frac{1}{1}\right) - \left(-a - \frac{1}{a}\right) \\ &= a - 2 + \frac{1}{a}. \end{aligned}$$

As you can see, when a approaches 0 from the positive real numbers, the value of $a - 2 + 1/a$ becomes huge. More precisely, we can write

$$\lim_{a \to 0^+} \left(a - 2 + \frac{1}{a}\right) = \infty$$

but then that also means

$$\lim_{a\to 0^+} \int_a^1 \left(-1 + \frac{1}{x^2}\right)\, dx = \infty$$

and, as it turns out, the same is true for the integral from -1 to 0 as well. Therefore, the original integral is equal to $\infty + \infty = \infty$.

1.12.7. The Assume Command, and Integrals

This example was identified by Joseph Loeffler, who brought it to my attention. Let's say that you wanted to evaluate the integral

$$\int_1^x \frac{t^3 + 30}{t}\, dt$$

and that you accordingly typed the Sage commands

```
var('t')
integrate( (t^3+30)/t, t, 1, x)
```

(By the way, `integrate` is a synonym for `integral`.) As it turns out, you'd receive back a response that is a huge error message. We must remember that in Sage, the last lines of an error message are the ones that are important. The last lines are as follows:

```
ValueError: Computation failed since Maxima requested additional
constraints; using the 'assume' command before integral
evaluation *may* help (example of legal syntax is 'assume(x-1>0)',
see 'assume?' for more details)
Is x-1 positive, negative or zero?
```

The complaint here, sent to Sage from Maxima, is justified because the function $f(t) = (t^3 + 30)/t$ will explode at $t = 0$ because it attempts to divide by zero. The integral is therefore improper if 0 is found between 1 and x. However, if $x - 1 > 0$, or, in plain English, if $x > 1$, then that's clearly not going to happen. Therefore, Sage is very justified in asking us to explicitly state this assumption.

Let's follow Sage's recommended course of action by adding the `assume` command before our `integrate` command. We now have

```
var('t')
assume( x>1 )
integrate( (t^3+30)/t, t, 1, x)
```

which produces the response

```
1/3*x^3 + 30*log(x) - 1/3
```

as expected.

The `assume` command is rare enough that many seasoned Sage users do not know that it exists. It is unlikely to come up in your work. However, we'll see another example of it on page 89 in Subsection 1.12.12 in an integral related to optics, on page 262 in Subsection 4.19.2 in a summation, and on page 287 in Subsection 4.23.2 in an integral related to a Laplace Transform.

1.12.8. A Challenge: Computer-Assisted Integration by Partial Fractions

First, consider

$$f(x) = \frac{17{,}680}{x^5 - 7x^4 + 20x^3 - 40x^2 + 64x - 48}$$

and use Sage's `f(x).partial_fraction()` command to compute the partial fraction decomposition. Second, compute the rest of the integral of $f(x)$ by hand, using Sage's partial fraction decomposition as your starting point. Compare your final answer to Sage's symbolic computation of the indefinite integral.

1.12.9. Impossible Integrals and Their Numerical Approximations

The word "impossible" is a dangerous one in mathematics. Some integrals are famously impossible. The two most famous ones are the Fresnel Integral

$$\int_0^x \sin \frac{\pi t^2}{2}\, dt,$$

which is important in optics, and the Gaussian Integral

$$\int_0^y \frac{2}{\sqrt{\pi}} e^{(-x^2)}\, dx,$$

which is pivotally important in statistics (and probability). What do we mean by "impossible" here? There is no function built up of finitely many additions, subtractions, multiplications, divisions, roots, exponents, logarithms, trigonometric functions, and inverse trigonometric functions[16] which will have, as its derivative, either the Fresnel Integral or the Gaussian Integral. The theory behind this claim will be somewhat described in Subsection 1.12.10, starting on page 84, for those of my readers who earnestly crave theoretical discussions.

For now, temporarily accept my word that these integrals are impossible. Yet, in applied mathematics, we need to calculate these integrals! In particular, the Gaussian Integral is of paramount importance in statistics. So what is done is that the definite integral can be computed by dividing the interval into many, many tiny regions. Lots of narrow rectangles and trapezoids can be used to approximate the area between the curve and the x-axis. The numerical areas of those rectangles and trapezoids are added together to give a numerical approximation for the integral. (That's why it is called "numerical integration.") Each of these rectangles or trapezoids might well have a small amount of error, but when you add up all the areas, the hope is that the errors cancel out a bit. You probably used this technique during your calculus class at some point, or will at some future point.

[16] We should also include the hyperbolic cousins of the trigonometric functions and inverse trigonometric functions, which you might or might not have been taught about.

Even more sophisticated methods for numerical integration include using polynomials instead of trapezoids, and Sage uses extraordinarily intricate techniques to make the best approximation possible. These approximations are an old and deep subject, going back to Simpson's Rule, where Thomas Simpson (1710–1761) used tiny slices of parabolas instead of narrow trapezoids. This allowed him to get rather accurate approximations using far fewer computations than the trapezoidal rule would require. Since he was working in the first half of the 1700s, that's a major accomplishment. In fact, essentially the same technique was used by the astronomer Johannes Kepler (1571–1630), and accordingly what is called Simpson's Rule in the USA is called Kepler's Barrel Rule in Germany.

The above might have been a bit confusing, but the whole point of Sage is that it will worry about these details for you, so long as you know what you want from Sage.

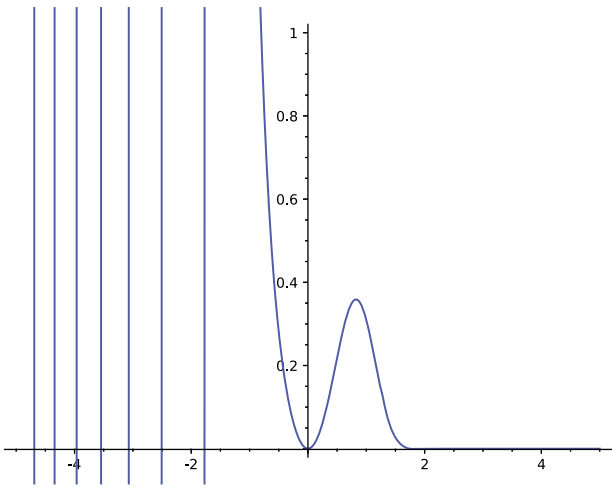
Let's take a specific example. Consider the function

$$f(x) = e^{-x^3} \sin(x^2),$$

which is an easily understood and continuous function. We can graph it with the commands

```
plot( exp( -x^3 ) * sin(x^2), -5, 5, ymin = 0, ymax=1 )
```

We obtain the following plot:



This function $f(x)$ seems intriguing, but computing its integral is rather frustrating. I cannot solve that integral with my pencil, and I am willing to wager that you cannot either. Try it if you like.

While it is the case that there is some function $g(x)$, somewhere, which has $f(x) = e^{-x^3} \sin(x^2)$ as its derivative—a logical consequence of some advanced real analysis and the fact that f is continuous—the practicalities of the matter are that there is no way to write down $g(x)$ as a finite formula. To be really precise, there is no function $g(x)$, built up of finitely many additions, subtractions, multiplications, divisions, roots, exponents, logarithms,

trigonometric functions, and inverse trigonometric functions (as well as their hyperbolic cousins, which you might or might not have been taught about), which will have $g'(x) = f(x)$.

In other words, $f(x) = e^{-x^3}\sin(x^2)$ is integrable, but the integral cannot be written in terms of common (and not so common) functions. Instead, we have to be content with numerical approximations, a topic that will be explained in Subsection 1.12.11, starting on page 86. Alternatively, one could produce a $g(x)$ made with an infinite series in Big-Sigma notation that would have $g'(x) = f(x)$, but we will not describe that here.

Returning now to Sage, since

$$\int e^{-x^3}\sin(x^2)dx$$

does not have a nice expression, then when you type

```
integral( exp(-x^3)*sin(x^2),x)
```

Sage replies not with an answer, but instead with

```
integrate(e^(-x^3)*sin(x^2), x)
```

which represents an admission of defeat. However, have no fear, because we can calculate numerical answers approximately—with very high accuracy—and we'll learn about that momentarily.

Before we do, for those of my readers that crave further specification on certain aspects of impossible integrals, we will have a theoretical interlude. Feel free to skip it and jump to Subsection 1.12.11, starting on page 86.

1.12.10. A Theoretical Interlude, Part Two

Once again, we come to a subsection that is absolutely worth reading, but depending on your mathematical background, you might find it rather hard to understand. Rest assured that you can understand essentially everything else in this book without understanding anything from this subsection at all.

In Subsection 1.9.4, starting on page 68, we discussed that it is not possible to find an exact formula for the roots of $x^5 - x + 1$ (where the formula is not infinitely long, and is composed of additions, subtractions, multiplications, divisions, integer exponents, and the extraction of square roots, cube roots, and higher roots), nor for many other polynomials of degree five and higher. We must be satisfied with mere numerical approximations for the roots of those polynomials. Similarly, some integrals are also impossible to solve exactly and have to be solved numerically.

We must be very careful to be precise about what we mean in this case of impossible integrals, just as we had to be when discussing impossible quintic polynomials. Let $f(x) = 4x^3 - 4x$. When we say that

$$\int 4x^3 - 4x\,dx = x^4 - 2x^2 + C,$$

what we really mean is that there exists an infinite family of functions $g(x)$, one for every real number C, and that once you pick your favorite real number to play the role of C, your specific choice of function for $g(x)$ has the property that

$$g'(x) = f(x) = 4x^3 - 4x.$$

Now let $f(x) = \sin(\pi x^2)$. Equivalently, when we say that

$$\int \sin\left(\pi x^2\right)\ dx$$

is an impossible integral, what we mean is that there is no way to write down a formula for any $g(x)$, such that $g'(x) = f(x)$ and where the formula is built up of finitely many additions, subtractions, multiplications, divisions, roots, exponents, logarithms, trigonometric functions, and inverse trigonometric functions (as well as their hyperbolic cousins, which you might or might not have been taught about).

This is really shocking, because we can prove that such a $g(x)$ does exist, even though we cannot write down a formula for it (where the formula is built up of finitely many additions, subtractions, multiplications, divisions, roots, exponents, logarithms, trigonometric functions, inverse trigonometric functions, hyperbolic trigonometric functions, and inverse hyperbolic trigonometric functions). We can even prove that this $g(x)$ is continuous, differentiable, twice differentiable, thrice differentiable, ..., and therefore smooth.

Another example is the very integral that is the core of calculus-based statistics. Namely, I am speaking of

$$\frac{1}{\sigma\sqrt{2\pi}} \int_a^b e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}\ dx,$$

whose numerical value is the probability that a normal random variable z, with mean μ and standard deviation σ, will fall into the interval $a < z < b$, provided that $a < b$. (Here "normal" means that the random variable follows the Gaussian distribution.) That's why there are large numerical tables of values for this definite integral in the back of nearly all introductory (and intermediate) statistics textbooks, except some that are focused only on performing statistics using computers.

Statistics is a common subject these days, taken by an enormous number of university students each year. Yet, in my experience in the USA, most of those students are either entirely unaware, or have forgotten, that statistics is based on an integral that is impossible. For example, anyone who has had any statistics at all has heard of the 68-95-99.7 rule, but I wonder how many know where those numbers came from. Those numbers come from

these integrals:

$$\begin{aligned} 0.682689\ldots &= \frac{1}{\sqrt{2\pi}}\int_{-1}^{1} e^{-x^2/2}\,dx, \\ 0.954499\ldots &= \frac{1}{\sqrt{2\pi}}\int_{-2}^{2} e^{-x^2/2}\,dx, \\ 0.997300\ldots &= \frac{1}{\sqrt{2\pi}}\int_{-3}^{3} e^{-x^2/2}\,dx. \end{aligned}$$

By the way, you can compute those yourself using Sage, in one line:

```
N( (1/sqrt(2*pi))*integral( e^(-x^2/2), x, -3, 3 ) )
```

Two more impossible integrals are

$$\int \frac{\sin x}{x}\,dx \quad \text{as well as} \quad \int x^x\,dx.$$

The theorem that helps mathematicians detect impossible integrals is called Liouville's Theorem, which is often taught either in an advanced course on mathematical analysis or alongside differential Galois theory, a somewhat obscure extension of Galois theory. To put matters in perspective, some universities in the USA that offer PhD programs in mathematics do not have a regularly offered course including differential Galois theory. Yet, the vast majority do have courses where Galois theory itself is taught, usually as a 6–10 week component of a course in *Abstract Algebra*, or sometimes as a course in its own right. For example, I have never had even one hour of instruction on differential Galois theory, but I know the "traditional" Galois theory fairly well.

It is really difficult to perform an internet search for Liouville's Theorem. That's because there are actually eight distinct theorems, depending on how[17] you count, in completely separate branches of mathematics, called Liouville's Theorem. Joseph Liouville (1809–1882) clearly knew several different branches of mathematics rather well, which is extremely humbling for the rest of us.

1.12.11. Numerical Integration in Sage

A nefarious integral, famous among calculus teachers, is

$$\int x^{20}e^x\,dx,$$

where the 20 could be any decently large number. This comes up in explaining what Euler's gamma function is, but we aren't too concerned with that right now. In any case, we could just type

```
integral( (x^20)*(e^x), x )
```

[17] I count complex analysis, conformal mappings, differential Galois theory, differential geometry, Hamiltonian mechanics, harmonic functions, linear systems of differential equations, and his theorem that there exist real numbers that are not algebraic numbers.

but then we get back

```
(x^20 - 20*x^19 + 380*x^18 - 6840*x^17 + 116280*x^16 - 1860480*x^15
+27907200*x^14 - 390700800*x^13 + 5079110400*x^12 - 60949324800*x^11
+ 670442572800*x^10 - 6704425728000*x^9 + 60339831552000*x^8
- 482718652416000*x^7 + 3379030566912000*x^6 - 20274183401472000*x^5
+ 101370917007360000*x^4 - 405483668029440000*x^3
+ 1216451004088320000*x^2 - 2432902008176640000*x
+ 2432902008176640000)*e^x
```

which is large enough that it isn't clear how to get an idea of what that means. If, instead, we had certain bounds, such as to calculate

$$\int_2^3 x^{20} e^x \, dx,$$

then we would type

```
integral( (x^20)*(e^x), x, 2, 3 )
```

and get instead

```
121127059051462881*e^3 - 329257482363600896*e^2
```

which is getting toward an answer! If you really want a numerical answer, then you should use our old trick, the command `N()`, and type

```
N( integral( (x^20)*(e^x), x, 2, 3 ) )
```

which would return the number

```
8.79797452800000e9
```

and that means $8.79797452800000 \times 10^9$. This highlights the differences and similarities of the various types of answers. If you've forgotten about `N()`, see Subsection 1.2.1, starting on page 7.

By the way, that number which we just calculated looks like an integer because it ends with 528 when you carry out the 10^9 part—but when we look at the exact answer, we can see that *no*, it is not an integer. We know that because e^2 and e^3 are involved and e is an irrational number.

We can confirm that remark in the following way

```
N( integral( (x^20)*(e^x), x, 2, 3 ), digits = 100 )
```

and we see that the answer is not actually an integer. It merely looked like one before we added the `digits = 100` because of rounding error. Whenever dealing with computations, it is critical to remember what is an approximation and what is exact. Irrational numbers can be expressed exactly on occasion (as we just did here, using e^2 and e^3) but that is the exception and not the rule. In any case, this paragraph has nothing to do with Sage—therefore we shall now return to the topic at hand.

Speaking of approximations, you can also jump to the numerical integral very rapidly. I'm sure we can both calculate

$$\int_0^1 (x^3 - x) \, dx = \left(\frac{1}{4}x^4 - \frac{1}{2}x^2 \right) \Big|_0^1 = \left(\frac{1}{4}1^4 - \frac{1}{2}1^2 \right) - \left(\frac{1}{4}0^4 - \frac{1}{2}0^2 \right) = \frac{1}{4} - \frac{1}{2} = -\frac{1}{4}.$$

However, if you wanted to ask Sage to calculate that integral numerically, then you must type

```
numerical_integral( x^3 - x, 0, 1)
```

which will output

```
(-0.24999999999999997, 2.775557561562891e-15)
```

where the first number is the best guess Sage has for the answer, while the second number is the uncertainty. In this case, the uncertainty is 2.77 quadrillionths—which is very impressive.

Of course, Sage can do this integral exactly as well, using the commands we learned in Subsection 1.12.4, starting on page 78. What is useful about numerical integration is the cases when you cannot do the indefinite integral, because like the Fresnel Integral (mentioned on page 82 in Subsection 1.12.9) it cannot be written but you can find good numerical estimates. Consider again

$$\int_1^3 e^{-x^3} \sin(x^2)\, dx$$

but now with bounds, so that it can be done numerically. We would type

```
numerical_integral( exp(-x^3)*sin(x^2),1,3)
```

and that returns back

```
(0.077997011262087815, 8.6840496851995037e-16)
```

with (as before) the first number being the answer, and the second the uncertainty. In this case, the uncertainty is 868 quintillionths—which is very impressive. Another way to say it is that the uncertainty is (1/1,152) trillionths.

1.12.12. The Function erf and Integrals

As I mentioned earlier, among mathematicians and especially among statisticians, the integral

$$\int_0^y \frac{2}{\sqrt{\pi}} e^{(-x^2)}\, dx = \text{erf}(y)$$

is extraordinarily important. This integral is very similar to the Gaussian Integral that I told you about earlier. You might remember that I said that the integral is impossible. It is so important that it has a name, and that name is `erf`. While being friendly and pronounceable, `erf` stands for the "Error Function."

In a literal sense, `erf(2)` can be thought of as saying "go look up the value for $-2 < z < 2$ in the data table in the back of some statistics textbook." Specifically `erf(x)` is the probability that a normal random variable z, governed by the Gaussian distribution with mean $\mu = 0$ and standard deviation $\sigma = 1$, will fall into the interval $-x < z < x$. That makes it a perfectly valid function, even if it must be computed numerically and not exactly.

Thus if you type into Sage

```
integral( (2/sqrt(pi)) * exp(-x^2), x, -oo, 2)
```

it will respond

```
(sqrt(pi)*erf(2) + sqrt(pi))/sqrt(pi)
```

which is correct because

$$\begin{aligned}\int_{-\infty}^{2} \frac{2}{\sqrt{\pi}} e^{(-x^2)}\, dx &= \int_{-\infty}^{0} \frac{2}{\sqrt{\pi}} e^{(-x^2)}\, dx + \int_{0}^{2} \frac{2}{\sqrt{\pi}} e^{(-x^2)}\, dx \\ &= \int_{0}^{\infty} \frac{2}{\sqrt{\pi}} e^{(-x^2)}\, dx + \int_{0}^{2} \frac{2}{\sqrt{\pi}} e^{(-x^2)}\, dx \\ &= 1 + \operatorname{erf}(2) \\ &= \frac{\sqrt{\pi}}{\sqrt{\pi}}(1 + \operatorname{erf}(2)) = \frac{\sqrt{\pi}\operatorname{erf}(2) + \sqrt{\pi}}{\sqrt{\pi}}.\end{aligned}$$

Perhaps you might find that $1 + \operatorname{erf}(2)$ is a more compact and simpler answer. I would be inclined to agree. I'm not entirely sure why Sage does not simplify its final answer further.

However, it is always possible to ask Sage for a simplification of the result, once it has been computed, using the `full_simplify()` command, also called `simplify_full()`. We can type

```
my_integral=integral( (2 / sqrt(pi) ) * exp(-x^2), x, -oo, 2 )
my_integral.full_simplify()
```

which will give us the $\operatorname{erf}(2) + 1$ that we expected.

In any case, the point is that there are some functions out there which do not have integrals writable using elementary functions, but which Sage calculates anyway using the `erf` function.

An Unexpected Connection to the Fresnel Integral:

It would be hard to imagine what it would mean for `erf` to be applied to a complex/imaginary number, but it can be done. If you wrote down an infinite series for `erf`, then it could be computed with a complex/imaginary input. As it turns out, `erf` is computed with a technique called "quadrature," but we cannot go into that now.

The Fresnel Integral, mentioned in Subsection 1.12.9, starting on page 82, which is important to optics, can be evaluated symbolically in terms of `erf`, but with a complex/imaginary input. This does not contradict the designation of this integral as impossible, because we did not include `erf` in our shopping list of acceptable ingredients.

Let's see this in action. To evaluate

$$\int_0^x \sin \frac{\pi t^2}{2} dt$$

we type in

```
var('t')
assume(x>0)
my_integral = integral( sin(pi*t^2/2), t, 0, x)
my_integral.simplify_full()
```

and we receive back

```
(1/4*I + 1/4)*erf(sqrt(1/2*I*pi)*x)
        - (1/4*I - 1/4)*erf(sqrt(-1/2*I*pi)*x)
```

which means

$$\left(\frac{1}{4}i+\frac{1}{4}\right)\,\mathrm{erf}\left(x\sqrt{\frac{1}{2}i\,\pi}\right)-\left(\frac{1}{4}i-\frac{1}{4}\right)\,\mathrm{erf}\left(x\sqrt{-\frac{1}{2}i\,\pi}\right)$$

and as you can see, we are giving `erf` a purely imaginary input. The idea of a probability that is an imaginary number makes me a bit queazy.

In any case, if we were to admit `erf` to our shopping list of acceptable ingredients, then both the Gaussian Integral from probability and statistics, as well as the Fresnel Integral from optics, would cease to be impossible integrals.

1.12.13. A Major Bug in Sage's Integral Command

On May 4, 2015, while teaching Math-154: *Calculus II* at UW–Stout, a course mostly about the integral calculus meant for engineering and science majors, I found a bug in how Sage computes integrals. This bug can come up in a broad set of situations, including arc-length integrals and any integral with a square root inside. An easy example is

$$\int_{\pi/4}^{3\pi/4}\sqrt{(\cot x)^2}\;dx.$$

Directly attempting to compute the integral exactly should be done with the very easy-to-understand code

```
integral( sqrt( cot(x)^2 ), x, pi/4, 3*pi/4 )
```

but Sage returns an answer of zero. One can compute this integral manually. My pencil produces an answer of

$$\log_e 2=\ln 2=0.693147180\ldots$$

after a bit of work.

Numerical methods give us some confirmation. As we saw in Subsection 1.12.11, starting on page 86, the output of the `numerical_integral` command is an ordered pair. The first number is the answer, and the second number is the uncertainty. The command

```
numerical_integral( sqrt( cot(x)^2 ), pi/4, 3*pi/4 )
```

gives an answer which is extremely close to `N(ln(2))`, which I know (from my hand calculations) is correct.

I think we can all agree that

$$\sqrt{(\cot x)^2} = |\cot x| \neq \cot x$$

because, in general,

$$\sqrt{(f(x))^2} = |f(x)| \neq f(x)$$

as we saw in the challenge on page 50 in Subsection 1.6.7.

However, I eventually figured out that either Sage, Maxima, or the interface between Sage and Maxima is causing the (incorrect) substitution of $\sqrt{(\cot x)^2}$ with $\cot x$, which is clearly wrong. Just plug in $x = -\pi/4$, and see that $\cot x = -1$ but

$$\sqrt{\left(\cot\left(-\frac{\pi}{4}\right)\right)^2} = \sqrt{(-1)^2} = \sqrt{1} = 1.$$

Next, using the correct substitution $\sqrt{(\cot x)^2} = |\cot x|$ manually, we can ask

```
numerical_integral( abs( cot(x) ), pi/4, 3*pi/4 )
```

and get again an answer which is extremely close to `N(ln(2))`.

However, if we use the (incorrect) substitution $\sqrt{(\cot x)^2} = \cot x$ manually, then we type

```
numerical_integral( cot(x), pi/4, 3*pi/4 )
```

and the result is extremely close to zero, matching the symbolic integral command's incorrect answer.

You might wonder how often an integral like this might occur. There are lots of applications where one takes the integral of a square root of either a square or a sum of squares. These come from the distance formula. For example, if $y = f(x)$, then the arc length along $f(x)$ from x_1 to x_2 is given by

$$\int_{x_1}^{x_2} \sqrt{1 + (f'(x))^2}\ dx.$$

(Some of my readers might not know what a parametric curve is, and they should skip to the last paragraph of this subsection.) For a parametric curve given by $x(t)$ and $y(t)$, the arc length from t_1 to t_2 is given by

$$\int_{t_1}^{t_2} \sqrt{(x'(t))^2 + (y'(t))^2}\ dt$$

and the particular problem I was solving that day was to compute the arc length of a parametric function given by

$$x(t) = \cos t + \log_e(\tan t/2), \quad y(t) = \sin t, \quad \pi/4 < t < 3\pi/4.$$

Computing $x'(t)$ is a non-trivial but healthy exercise in calculus and trigonometry, given below:

$$\begin{aligned} x'(t) &= -\sin t + \frac{d/dt\ \tan t/2}{\tan t/2} = -\sin t + \frac{(\sec t/2)^2(1/2)}{\tan t/2} \\ &= -\sin t + \frac{(\sec t/2)^2}{2\tan t/2} \cdot \frac{\cos t/2}{\cos t/2} = -\sin t + \frac{\sec t/2}{2\sin t/2} \\ &= -\sin t + \frac{1}{\cos t/2} \cdot \frac{1}{2\sin t/2} = -\sin t + \frac{1}{2(\sin t/2)(\cos t/2)} \\ &= -\sin t + \frac{1}{\sin t} = -\sin t + \csc t. \end{aligned}$$

After that, it is easy to compute

$$\begin{aligned} \big(x'(t)\big)^2 &= (-\sin t + \csc t)^2 \\ &= (\sin t)^2 + 2(-\sin t)(\csc t) + (\csc t)^2 \\ &= (\sin t)^2 - 2 + (\csc t)^2 \end{aligned}$$

and since

$$y'(t) = \cos t$$

clearly

$$\big(y'(t)\big)^2 = (\cos t)^2$$

implying that

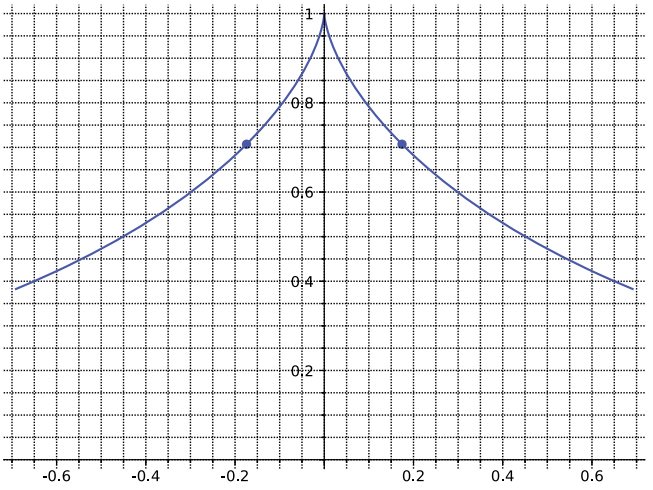
$$\begin{aligned} \big(x'(t)\big)^2 + \big(y'(t)\big)^2 &= (\sin t)^2 - 2 + (\csc t)^2 + (\cos t)^2 \\ &= (\sin t)^2 + (\cos t)^2 - 2 + (\csc t)^2 \\ &= -1 + (\csc t)^2 = (\cot t)^2 \end{aligned}$$

which is why I needed to compute

$$\int_{\pi/4}^{3\pi/4} \sqrt{(\cot t)^2}\ dt$$

in the first place.

If you are curious, a plot of the function is given below. As you can see, the distance between the two points clearly isn't zero, neither traveling along the curve to obtain arc length, nor by taking "a direct flight."



Even the following Sage program, ideal for explaining the process, gave an answer extremely close to zero:

```
var('t')
x(t) = cos(t) + ln( tan(t/2) )
y(t) = sin(t)
x_p(t) = diff( x(t), t )
y_p(t) = diff( y(t), t )
g(t) = sqrt( (x_p(t))^2 + (y_p(t))^2 )
my_integral = integral( g(t), t, pi/4, 3*pi/4 )
print( N( my_integral ) )
print( my_integral.simplify_full() )
```

The moral of the story is that for arc lengths and any integrals with a square root in them, it is better to use `numerical_integral` than to compute the integral symbolically. Moreover, sometimes there is much to be gained by examining a plot.

1.13. Sharing the Results of Your Work

Scientific collaboration is one of the great joys of working in a STEM[18] subject. I hope that you will have frequent opportunities to share not only your ideas, but also the hardcore mathematical and quantitative details of your work, with others. Since Sage has been developed by hundreds of people scattered across the globe, it is unsurprising that there are many features built into Sage to enable sharing.

In this section, I'll describe ten effective ways of sharing the results of your work in Sage. These can be great for scientific collaboration, submitting work to a faculty member, getting help from a friend, or just explaining a math concept to someone. My two most favorite methods are accessed via the "Share" button on the SageMathCell screen, so we'll start there.

[18]STEM: Science, Technology, Engineering, and Mathematics.

1.13.1. Two Ways of Sharing Your Work with URLs

When you have work in the SageMathCell window, you can click on the "Share" button. Then you will see options for making a short temporary link, a permalink, or a "2D barcode," also called a "QR code."

The permalink is my favorite. It will generate an enormous URL and prove to you that the new URL works by reloading the page with that URL. Then you should highlight that URL from the web browser, from the spot where you would normally type `https://sagecell.sagemath.org/`, taking great care to make sure that you've highlighted the whole thing. This permalink can now be used in documents, in emails, in chats, or even on Facebook or LinkedIn. I frequently use these URLs in the course management systems that my university provides (D2L until 2019, and later, Canvas). The permalink actually encodes the Sage code in the window. Thus your Sage code is not stored anywhere, and therefore it cannot be lost, deleted, or destroyed. As long as you retain a complete and unmodified permalink, then it will always work, even years later.

As I mentioned before, the change that occurred during December 2019 and January 2020, bringing Sage from Python 2 to Python 3, rendered inoperable about 2/3 of the examples in the first edition of this book, mostly but not entirely due to the changes in the `print` command's syntax. Nonetheless, even the oldest permalinks, which I've accumulated over the many years that I've been using Sage, still produced—even in the summer of 2020—exactly the code that was typed when I made the permalink. All I had to do was modify the syntax of the `print` command, modernizing it by adding parentheses. Only in rare cases were additional steps needed, and even then only a few keystrokes.

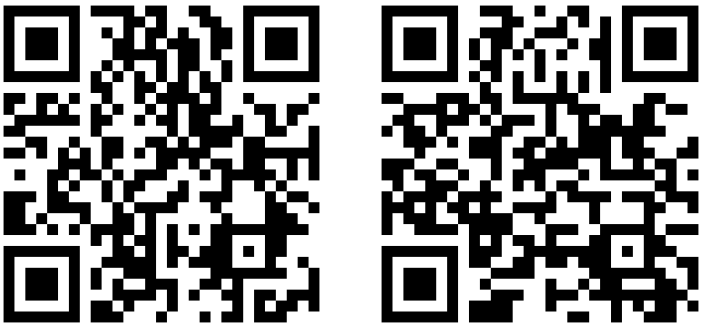
The permalink is often very long and ugly. This should not be surprising because it actually encodes your entire block of Sage code. In fact, the permalink is so long that it can cause problems in some extremely old web browsers or email processors. In almost all cases, it is longer than 280 characters; therefore it cannot be tweeted on Twitter. The short temporary link, on the other hand, is shorter and less intimidating to a human. Being shorter, it is less likely to be truncated in transit when sent by various means.

However, the short temporary link does not contain a complete copy of your code. Instead, your code is stored on the server, and the URL retrieves it. There is no guarantee for how long the short temporary link will work. Perhaps a day, perhaps a year, or perhaps a decade, but perhaps only an hour. In my experience, they often work several days after being generated. Overall, I'd have to say that if the length and ugliness are not a problem, then the permalink should always be used and the short temporary link should never be used.

1.13.2. Sharing Your Work with a QR Code (2D Barcode)

The third method I'd like to share with you is the large 2D barcode produced when you click "Share." This actually encodes the permalink and thus will work with all smartphones that can read those 2D barcodes. Most phones will require you to install an app for the smartphone to be able to handle those barcodes, which are sometimes called "QR codes," "Quick Response Codes," or "matrix barcodes." On my phone, the app[19] was called "QR Code Reader" and was free. If you photograph the barcode using one of those apps and the camera inside your smartphone, then it should open up a web browser and present the same results as the permalink will. You don't even need to click "Evaluate"!

Here are two sample 2D barcodes, from making volumes of revolution in *Calculus II*, as an application of integration. I've decided to revolve only 75% of the way around, thus 270°, so that the inside is visible. You can use your fingers to move the object around, and you can also zoom in or out with the usual pinching movements.



1.13.3. Sharing Sage Plots as PNGs (Image Files)

The fourth method is when you've made a plot (as we learned in Subsection 1.4.1, starting on page 23) and you'd like to include this image in any sort of document or email it to someone. Just save the image file from your web browser—in the same way that you'd save any other image. In most browsers, this is a right-click followed by selecting "Save Image As..." and entering in a filename at the appropriate spot. This image will be in the "portable network graphic" or `*.png` format. It will work in nearly all programs that allow the importing of an image.

1.13.4. Sharing EPS Files (Vector Graphics Images)

Instead of the `*.png` style images, some high-end mathematics and physics journals prefer "encapsulated postscript" or `*.eps` files. Those can be

[19] You really do need to use an app meant for 2D barcodes, however. Otherwise, if you use an ordinary camera app, then you just get a photograph of the QR code, which is rather unexciting.

generated easily and are our fifth method. Here is an example:

```
P = plot( x^3-x, -2, 2 )
P.save('myplot.eps')
```

which will give you a link that will start the downloading of `myplot.eps` when clicked upon. These are vector graphics, in the sense that the image is stored as a set of line segments, polygons, and circles. That's in contrast to pixel graphics, where the image starts out as a grid of pixels, and then one or more compression algorithms are applied. With pixel graphics, if you zoom in extremely far, then you can see the individual pixels, and that makes curves or diagonal lines look terrible. With vector graphics, you can zoom in arbitrarily far, and the image will be correct.

1.13.5. Sharing Your Code with Cut-and-Paste

The sixth method is just highlighting the Sage code, copying, and then pasting it into an email. Sometimes I will paste it into `CoCalc.com` or into my university's course management software, Canvas, but formerly D2L. Emailing code can save time, and it is easy to share code in this way.

I used this method of sharing frequently, during my first few years of using Sage. However, it has some major disadvantages. As you will learn in Chapter 5, indentation is extremely important in both Sage and Python code. It shows which commands are governed by other commands, such as `if`-then-`else` structures and structures called `for` loops and `while` loops. (Don't worry if you don't know what those are now. We'll cover them in tremendous detail in Chapter 5, starting on page 317.) Cutting and pasting code, especially via email, often will damage or destroy the indentation. After these modifications to the indentation, the code will usually not operate at all, or it will sometimes operate incorrectly.

Unrelated to indentation, in order to make your quotation marks (") and apostrophes (') more stylish, in some email programs they will be replaced with curvier or tilted versions of themselves (perhaps in the style of “ or ”), represented by something called Unicode. After such replacement, in most cases the Sage or Python code will generate error messages instead of operating correctly. By the way, it is not necessary to know what Unicode is, or how it works, to be proficient in Sage.

1.13.6. Sharing Your Work via Screenshot

Seventh, you can use screenshots. This method does not work well, but it is quick and thus convenient. It can be useful in some circumstances. Making screenshots is a good skill for everyone to have in situations unrelated to Sage, such as if someone threatens you during an internet chat, or if you're experiencing bizarre error messages from a game that is being beta tested.

The keyboard shortcuts chosen by operating-system designers change once in a while over the decades. However, what follows has been unchanged

throughout the first twenty years of the 21st century, so it probably will remain unchanged for the next few years.

- For Microsoft Windows, use the "Print Screen" button. On keyboards in the USA, it has `Prt Sc` above `Sys Rq` written on it and is usually in the top row of the keyboard. (My proofreader, Ryan Hornberger, reports that one can also press Shift + Windows Key + S instead of using the "Print Screen" button.) That makes a screenshot and copies it to the clipboard. However, it must be pasted somewhere, such as in a document, into an email, or into the Paint tool, formerly called paintbrush. To paste, use the same keystroke used in ordinary cut-and-paste actions.
- In Linux, the "Print Screen" button is also used (for most distributions). However, after pressing that button, it pops up a dialog box with several options.
- On the Apple Macintosh, command-shift-3 is the keystroke. I like to remember this as "command hash tag." (On keyboards outside of the USA, it is possible that the key sequence might be different.) After pressing this key combination, the screen shot is saved as an image file, in PNG format, to the desktop. The filename also contains a timestamp, which can be handy in situations where knowing the date and time matters. You can also use command-shift-4 to grab only a portion of the screen, or command-shift-5 to get a small menu of several choices.

The screenshot almost always shows the date and time it was made. In a class with 35 students, the instructor might see 8–12 students submit identical code, down to the spacing. It is particularly fascinating when all of those screenshots have exactly 6:23 PM on September 26 as the timestamp. What an amazing coincidence that each student chose to make their screenshot at exactly that specific moment in time.

It should be noted that many undergraduates, not knowing how to make screenshots, will instead photograph the screen of their laptop with their smartphone. The resulting image is of much lower quality, and working with such images is difficult. Sometimes if two or three such photos are needed, the student might accidentally email me four or five images. This results in me seeing whatever they might have been photographing with their smartphone prior to the Sage session in question.

A major drawback of screenshots is that you cannot cut-and-paste code from them. Screenshots are just images. Also, if there is a lot of code, then it won't fit in the SageMathCell window all at the same time. Of course, the screenshot will not contain any code that isn't being displayed at the moment it was taken.

1.13.7. Sharing Your Work via Hardcopy Print Out

The eighth method is to print a screenshot of the web-browser window while using SageMathCell. A few years ago, this was often how I asked students in my classes to submit a homework problem that was to be done in Sage. However, if there is a lot of code—more than will fit in the SageMathCell window—then some of it might not be visible. This flaw is also a problem with screenshots, but with hardcopy printouts, I think the flaw is somewhat less predictable.

If the size of the code isn't too large, then using a bit of scrolling around and resizing, you can often cause the entire Sage block of code and its output to be visible at the same time. Then you can use your browser's Print menu option (often under the "File" menu) to print the webpage just as you'd print any other webpage.

Like screenshots, the printout usually will include the date and time. This has advantages, especially when keeping track of research progress.

The disadvantages of printouts are obvious. First, it seems that the percentage of students who own a printer is decreasing every year, at least in the midwestern USA. Second, this does not work well with online courses. Third, it is a silly waste of paper, with environmental consequences. The next subsection will give us a better technique.

1.13.8. Sharing Your Work via PDF Files

The ninth method is to follow the steps for making a hardcopy printout while using SageMathCell, but instead of printing it as a physical paper document, to select "Save as a PDF" from the web browser's dialog box for printing. If your computer has n printers, often "Save as a PDF" is treated as the $(n+1)$th printer, or there might be a separate "PDF" pull-down selector in the dialog box for printing. This PDF file is essentially identical as a document to what you'd get from printing, except that it is an electronic copy, rather than a paper hardcopy.

This has the obvious advantage of saving paper, an important environmental concern. It also resolves the issue of low-income students who might not own a printer. However, the same problem comes up with large quantities of code that we saw with hardcopies. Not all of the code will be visible if there are too many lines. Overall, permalinks are the best choice. That's why I described them first.

1.13.9. Sharing Your Work via CoCalc.com

Our tenth method, and by far the best, is the website `CoCalc.com`. It has an enormous variety of features to help Sage users—so many, in fact, that it is difficult to summarize them here. The Sage capabilities of `CoCalc.com` make your work more permanent than SageMathCell, in the sense that your work is saved forever, or until you delete it.

- Some nice features on the research or "advanced coursework" side include...
  - You can organize your work into projects, each with a different set of collaborators. A project is like a directory (or folder) on a computer, but any collaborator can create, read, edit, and delete as needed. If you enjoy being organized, you can make subdirectories and sub-subdirectories—as many levels as you might like.
  - All of your Sage code, and the output generated, can be kept forever. This includes any data sets or images.
  - Using a feature called "time machine," you can see every version of every file that ever existed in your project. It operates as a scroll bar, and you can travel through a document's past to any arbitrary point.
  - There is a live chat system, for chatting with collaborators. You can use LaTeX (a mathematical document-preparation language) to add formulas to your remarks during chat.
  - There is support for an enormous number of programming languages, not just Sage and Python. At this point, I think that every language that I've ever heard of is supported.
- Some nice features on the teaching side include...
  - You can generate a handout in Sage, and then with one click, you can send it to all of your students. They each get a copy, which they can modify.
  - You can generate an assignment, which each student gets a copy of. They work on it, and then `CoCalc.com` will manage the collection of each student's assignment. While the submissions have to be graded by a human, there are features to help record all the grades.
  - It is easy for instructors (or teaching assistants) to look at a student's worksheets, to give them help if they get stuck.
  - Even though most students probably do not know LaTeX, if the instructor knows it, then the chatroom feature is a great way of holding office hours.

You can learn about all of these features on the `CoCalc.com` website.

1.14. A Technicality about Functions

There is now only one concept remaining that should be clarified, and then you'll be ready to read about nearly any aspect of Sage. As I've mentioned before, once you've completed Chapter 1, you can read any of the later chapters of this book in any order (except Chapter 6 should follow after Chapter 5). You can also read any section from Chapter 3 or from Chapter 4 at any moment after finishing Chapter 1. Similarly, the sections of Chapter

2 can be done in any order, but they have prerequisites in some cases, which are clearly noted.

There is a technicality about how Sage treats functions that I would like to explain now. This was left out of the first edition of this book, and that omission caused some readers to get stuck on the relatively rare occasions when this technicality matters.

As a simple example, let's consider a polynomial. I'm also going to declare a dummy variable, `h`, to make a point. The code

```
var('h')
f(x) = x^2 - 5*x + 6
print( 'f(1) =', f(1) )
print( 'f(2) =', f(2) )
print( 'f(h) =', f(h) )
print( 'f(x+h) =', f(x+h) )
print( 'f(x) =', f(x) )
print( 'f itself =', f )
```

will produce the output

```
f(1) = 2
f(2) = 0
f(h) = h^2 - 5*h + 6
f(x+h) = (h + x)^2 - 5*h - 5*x + 6
f(x) = x^2 - 5*x + 6
f itself = x |--> x^2 - 5*x + 6
```

There's no surprise in the output of the first three `print` statements. We see that $f(1)$ is what you get when you evaluate the function f at $x = 1$, namely 2. Similarly, $f(2)$ and $f(h)$ are what you get when you evaluate the function f at $x = 2$ and at $x = h$. In the former case, you get 0, and in the latter case, you get $h^2 - 5h + 6$. The fourth `print` statement is a classic exercise at the beginning of calculus, where we want to know what we obtain when we plug in $x + h$ in place of x. Both at the start of calculus, and while using Sage, we write that as $f(x + h)$. William Stein, the creator of Sage, was careful to ensure that Sage syntax is close to (or even identical to) ordinary mathematical notation when possible.

In the fifth `print` statement, when we write $f(x)$ in Sage, we're literally asking Sage to tell us what we get when $x = x$, but figuratively we're asking for the formula to be used when computing the value of f at any specific x-value. Therefore, when we ask for $f(x)$, we get back the formula for $f(x)$ that we wrote when we defined $f(x)$.

It would be extremely reasonable for the reader to expect that `print( f )` and `print( f(x) )` should be synonymous. That's not actually true, however. It is necessary, once in a while, to be able to talk of f as a mapping. In this case, it is a moderately straightforward mapping, but more complex mappings are common in Sage. For example, we could easily define something that takes as input a vector and produces a matrix. Or we

could easily define something that takes as input a matrix and produces a scalar (a real number), such as the determinant.

The symbol `|-->` in Sage means "maps to" or "evaluates to." Therefore, when we ask for `print( f )` and receive back

```
x |--> x^2 - 5*x + 6
```

what we are seeing is notation that means f is a mapping that sends x to $x^2 - 5x + 6$. Whatever is to the left of `|-->` is the input of the mapping, and whatever is to the right of `|-->` is the output of the mapping.

This `|-->` symbol is meant to represent the mathematical notation

$$f\colon x \longmapsto f(x)$$

in general, or

$$f\colon x \longmapsto x^2 - 5x + 6$$

in this specific case. That mathematical notation is often used in advanced undergraduate courses or graduate-level courses. If you've never seen that notation before, then don't worry about it, because it is not necessary for understanding this subsection.

Some additional clarity can be gained by closely examining the following example:

```
f(x) = x^2 - 5*x + 6
g(x) = x^2 - 8*x + 15
print( 'Formal functions:' )
print( f + g )
print( f.factor() )
print( gcd(f, g) )
print( )
print( 'Values of functions:' )
print( f(x) + g(x) )
print( f(x).factor() )
print( gcd(f(x), g(x)) )
```

That code, which was kindly provided by Diego Sejas, produces this output:

```
Formal functions:
x |--> 2*x^2 - 13*x + 21
x |--> (x - 2)*(x - 3)
x |--> x - 3

Values of functions:
2*x^2 - 13*x + 21
(x - 2)*(x - 3)
x - 3
```

As you can see, if we give commands in terms of $f(x)$ and $g(x)$, then we get answers in that notation. Similarly, if we give commands in terms of f and g, what a mathematician would call formal mappings and what

a computer scientist calls function pointers, then we get answers in that notation.

Personally, I prefer the $f(x)$ notation, and I use that myself, including throughout the majority of this book. You should use whichever format you prefer. I will refer to both f and $f(x)$ in discussions with the word "function," as it is not practical to make a verbal distinction between these two essentially identical systems of notation.

We can replace `f.factor()` with the equivalent form `factor( f )`, in the same way as `f(x).factor()` can be replaced with `factor( f(x) )`.

You will see later that, due to syntax rules which Sage inherited from Python, it will be necessary to use f and not $f(x)$ when doing something called "passing a function as a variable to a subroutine," or equivalently "passing a function pointer to a subroutine." However, we will see that starting on page 338 in Subsection 5.2.5, so there's no reason for us to worry about that now.

Thank you for reading! I hope that you enjoy the rest of this book and your explorations of Sage.