MA241 Manual

Mathematical Computation and Modeling

MA241 Manual

Mathematical Computation and Modeling

Robert D. Poodiack
Norwich University

January 16, 2026

Contents

Part I

SageMath and Python Basics

Chapter 1

Acquiring access to SageMath

SageMath, a computer algebra system built on the Python programming language, is the software we'll use in this course. In this chapter, we get you started by showing you how to download SageMath to your computer or how to access SageMath without downloading it.

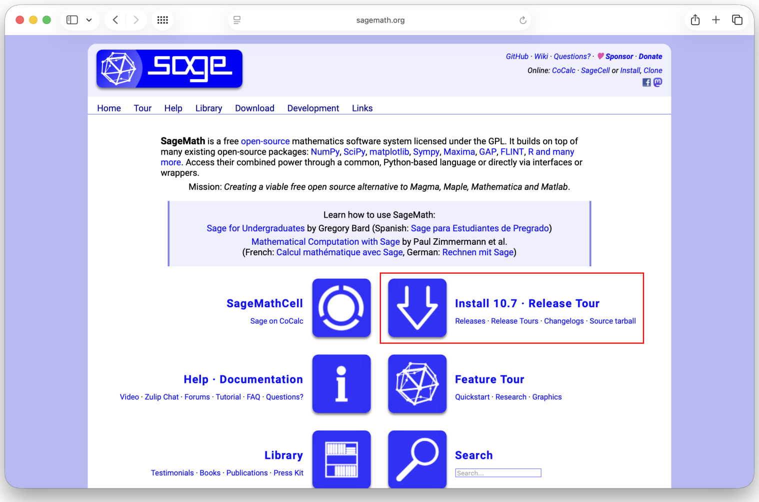
1.1 Downloading SageMath

SageMath is an open-source and free competitor to expensive computer algebra systems like Mathematica, Maple, and MATLAB. It is generally easier to use than a graphing calculator and much more powerful, combining ways to display your results to others with stunning graphics.

Working with SageMath can be done by downloading the software to your own laptop, or by using CoCalc, an online installation that can be used in any browser. There are advantages and disadvantages to each implementation, but both experiences are equivalent -- you get the same version of Sage and can use the same commands in either method.

What follows below are starter instructions for downloading SageMath to your Mac or a Windows PC. (There is an app for the iPad and the iPhone, but these simply call the SageMathCell server to do quick calculations. At this writing, there is not a full implementation of SageMath for the iPad or iPhone. On the iPad, you will need to use CoCalc.)

In any browser, navigate to http://www.sagemath.org.



sagemath.org

GitHub · Wiki · Questions? · Sponsor · Donate
Online: CoCalc · SageCell or Install, Clone

Home Tour Help Library Download Development Links

SageMath is a free open-source mathematics software system licensed under the GPL. It builds on top of many existing open-source packages: NumPy, SciPy, matplotlib, Sympy, Maxima, GAP, FLINT, R and many more. Access their combined power through a common, Python-based language or directly via interfaces or wrappers.

Mission: *Creating a viable free open source alternative to Magma, Maple, Mathematica and Matlab.*

Learn how to use SageMath:
Sage for Undergraduates by Gregory Bard (Spanish: Sage para Estudiantes de Pregrado)
Mathematical Computation with Sage by Paul Zimmermann et al.
(French: Calcul mathématique avec Sage, German: Rechnen mit Sage)

SageMathCell
Sage on CoCalc

Install 10.7 · Release Tour
Releases · Release Tours · Changelogs · Source tarball

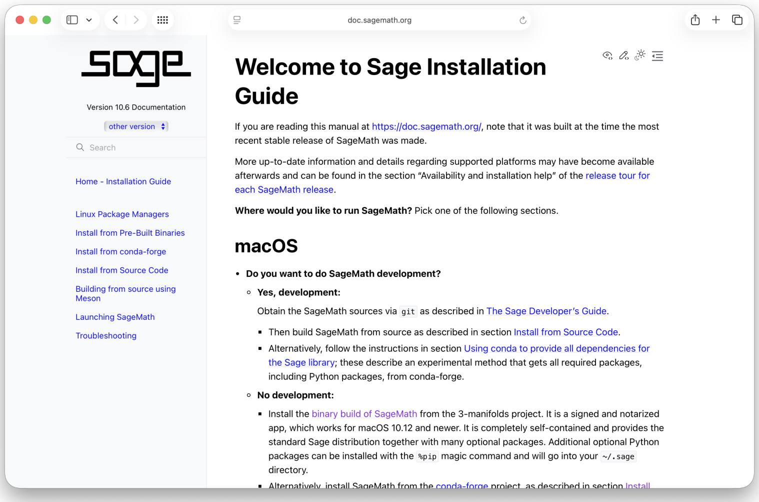
Help · Documentation
Video · Zulip Chat · Forums · Tutorial · FAQ · Questions?

Feature Tour
Quickstart · Research · Graphics

Library
Testimonials · Books · Publications · Press Kit

Search
Search...

There are six buttons in the center of the screen. Click on the downward arrow button in the upper right-hand corner. (It'll say "Install 10.7" -- or whatever the current version is.)



doc.sagemath.org

Version 10.6 Documentation
other version
Search

Home - Installation Guide

Linux Package Managers
Install from Pre-Built Binaries
Install from conda-forge
Install from Source Code
Building from source using Meson
Launching SageMath
Troubleshooting

Welcome to Sage Installation Guide

If you are reading this manual at https://doc.sagemath.org/, note that it was built at the time the most recent stable release of SageMath was made.

More up-to-date information and details regarding supported platforms may have become available afterwards and can be found in the section "Availability and installation help" of the release tour for each SageMath release.

Where would you like to run SageMath? Pick one of the following sections.

macOS

- **Do you want to do SageMath development?**
  - **Yes, development:**
    Obtain the SageMath sources via `git` as described in The Sage Developer's Guide.
    - Then build SageMath from source as described in section Install from Source Code.
    - Alternatively, follow the instructions in section Using conda to provide all dependencies for the Sage library; these describe an experimental method that gets all required packages, including Python packages, from conda-forge.
  - **No development:**
    - Install the binary build of SageMath from the 3-manifolds project. It is a signed and notarized app, which works for macOS 10.12 and newer. It is completely self-contained and provides the standard Sage distribution together with many optional packages. Additional optional Python packages can be installed with the `%pip` magic command and will go into your `~/.sage` directory.
    - Alternatively, install SageMath from the conda-forge project, as described in section Install

NOTE: If you have installed SageMath on MacOS, you should move the icon to start SageMath to the Applications folder. If you have installed SageMath on a Windows machine, you should absolutely create a shortcut for SageMath that lets you start the Jupyter or JupyterLab server in one click.

1.2 Using Sage without downloading

As an alternative, you can work with Sage through a cloud-based installation called CoCalc, available through http://www.cocalc.com. Your instructor will have likely established accounts for your entire class, and you will have received an invitation to join via email. The pro for using CoCalc is that you don't have to go through an installation process. The con is that CoCalc purposely runs free accounts on a slow server, and it will almost always be slowest on the night before your homework's due date.

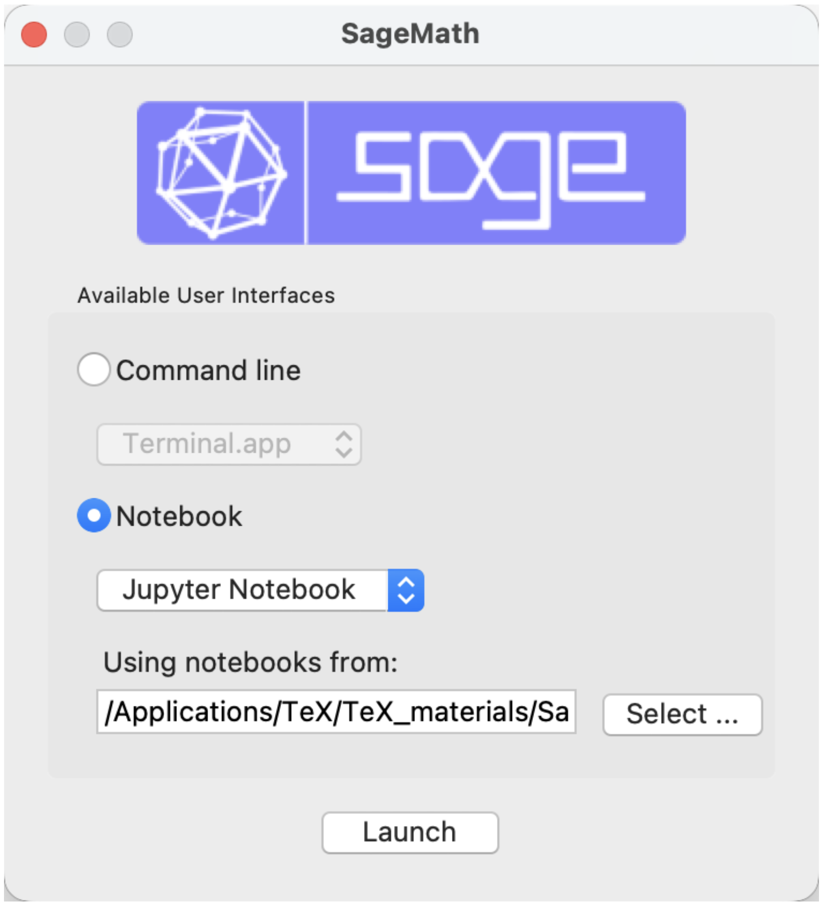
Chapter 2

Getting Started with SageMath

Now that you have access to SageMath, either by downloading it to your own computer or by accessing it in CoCalc, we'll start with some basic commands in Sage to get you on your way.

2.1 Starting up SageMath

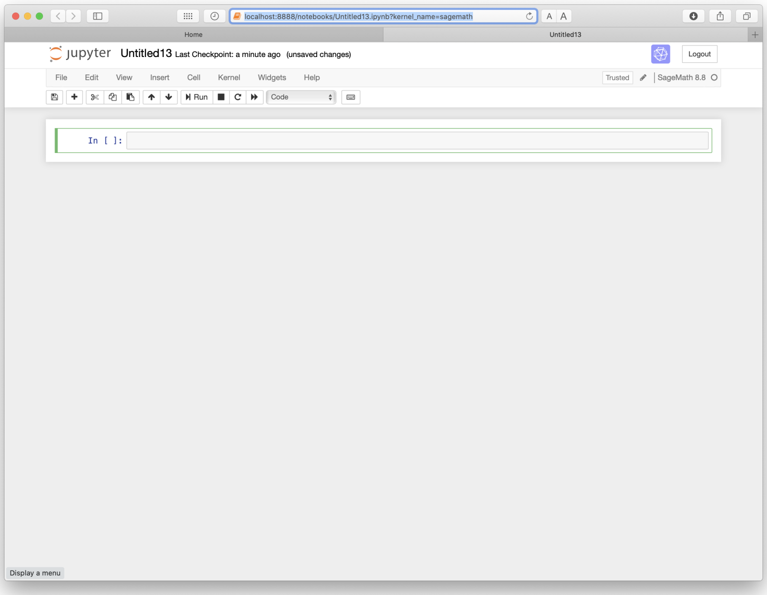
If you downloaded SageMath on to your own computer, you should be able to simply double-click on the Sage icon to start (SageMath-X-X in the Applications folder on the Mac, whatever you named your shortcut on Windows machines).



(Make sure on MacOS, you choose the Notebook option.) After a short pause, you should land on your default directory page. If you click the New pulldown menu in the upper right-hand corner of your screen, choose SageMath X.X for your current version.

If your instructor has given you access to CoCalc, you should be able to log in to your account and then choose New and Jupyter notebook.

In any case, you should be in a notebook environment that looks something like this:


localhost:8888/notebooks/Untitled13.ipynb?kernel_name=sagemath
Home
Untitled13
jupyter Untitled13 Last Checkpoint: a minute ago (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
SageMath 8.8
Run
Code
In [ ]:
Display a menu

From this point forward, you should follow along by typing given examples into your Jupyter notebook. (If you are reading along with the Web-based version of this book, you can hit the Evaluate button after each example.)

2.2 Working with Jupyter notebooks

While SageMath can be used within a terminal environment, using SageMath in a Jupyter notebook leads to much more robust and readable sessions. In a Jupyter notebook, we can enter explanatory text to go with the SageMath code using HTML commands to format. In this section, we will give some examples of coding that will allow us make our notebooks look exactly as we want them to look.

2.2.1 Getting started with Jupyter

When you open a new notebook in either SageMath or CoCalc, you'll have one of two similar but not identical experiences.

- In SageMath, you'll go up to the right-hand corner of your screen to the New button and choose SageMath 10.8 (or whatever version you have). A blank notebook with an indicator that you're using SageMath as the kernel should open.
- In CoCalc, you can click on the arrow to the right of the New button and choose "Jupyter notebook (.ipynb)". You'll be asked to name the file. After you do so, you should see the screen in Figure 2.2.1.

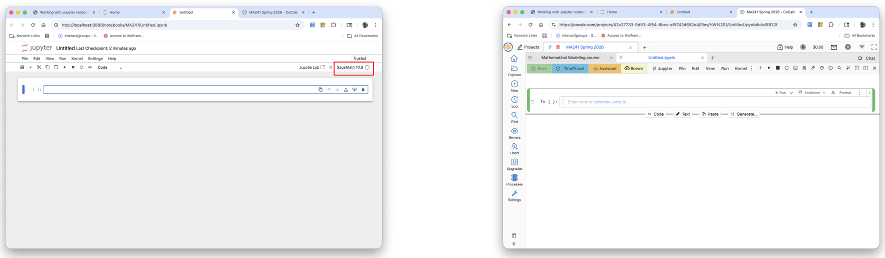


Figure 2.2.1 A blank notebook in SageMath (left) and in CoCalc (right).

You may have to go to the Kernel menu in CoCalc to select the most recent version of SageMath as the kernel.

2.2.2 Adding plain text in Jupyter notebooks

We assume that you have a blank Jupyter notebook open either in CoCalc or in SageMath itself.

We will introduce SageMath code in the next section. For now, we will look at ways of entering and formatting plain text in our notebook. This is useful for adding titles, section headings, and longer comments on your code.

Content in Jupyter notebooks is inserted via *cells*. In both SageMath and CoCalc, new cells are by default *Code* cells, the kind of cell where we would enter SageMath commands. To enter plain text, we need to have a *Markdown* cell.

In SageMath, we can click on the pulldown menu right and choose "Markdown" as the type of cell. (See Figure 2.2.2.) In CoCalc, we can either add a new cell by choosing to add a Text cell, then choosing Markdown. (see Figure 2.2.3) or clicking on the three dots at the right end of the cell and choosing Change Cell to Markdown. (See Figure 2.2.4.)

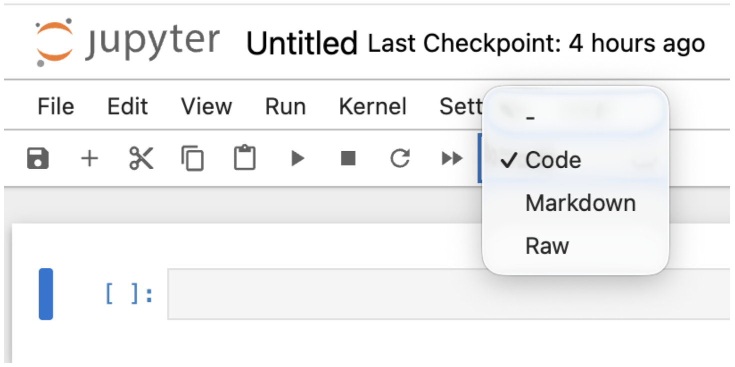


Figure 2.2.2 Changing the cell type to Markdown in SageMath

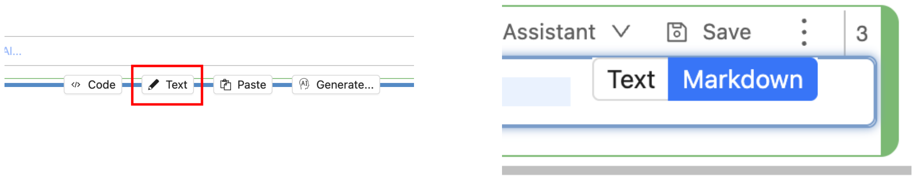


Figure 2.2.3 Adding a new Markdown cell in CoCalc

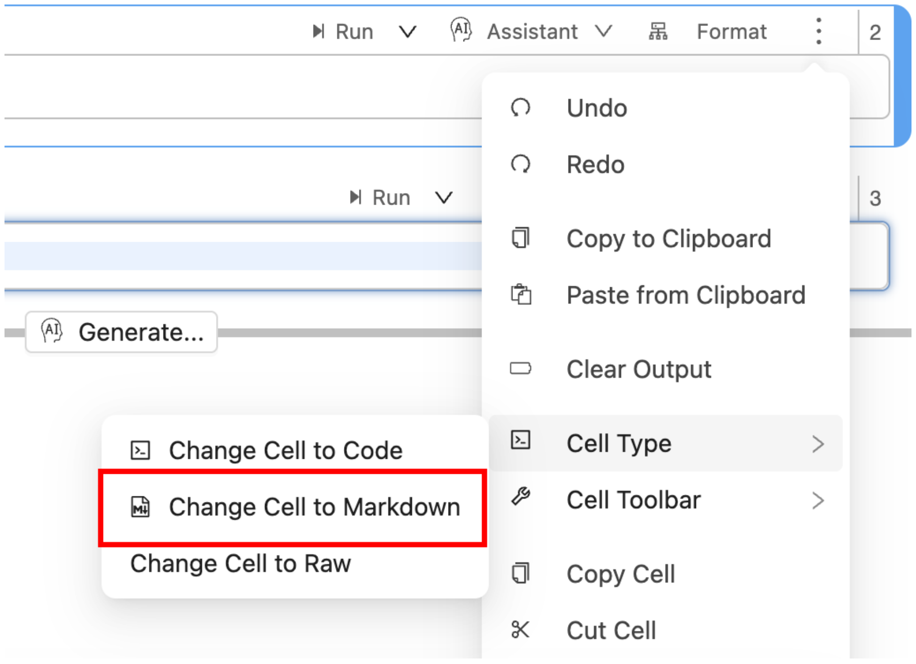


Figure 2.2.4 Changing a cell to Markdown in CoCalc

2.2.3 Formatting plain text

Here, we provide several examples of ways to enter and format plain text in Jupyter notebooks. Jupyter notebooks use HTML tags and commands to format text. We only need a few basic commands to make our notebooks look spiffy. Once our Markdown commands are entered, we hit `SHIFT-ENTER` to render the text.

Example 2.2.5 Entering plain text. Plain text in Jupyter notebooks must begin with a `<p>` tag (for *paragraph*) and end with a `</p>` tag. Each new paragraph must begin with `<p>` and end with `</p>`.

```
<p>This is a piece of plain text (not code).</p>
```

This is a piece of plain text (not code).

□

Example 2.2.6 Entering bold or italic text.

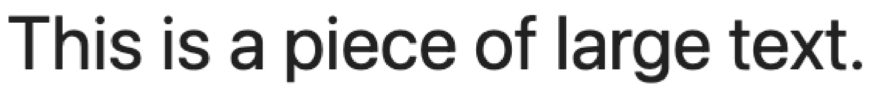
```
<p>This is a piece of <b>bold</b> text and this is a piece
of <i>italic</i> text.</p>
```

This is a piece of **bold** text and this is a piece of *italic* text.

□

Example 2.2.7 Entering large text. Here, we ask SageMath to render our text two sizes bigger than usual.

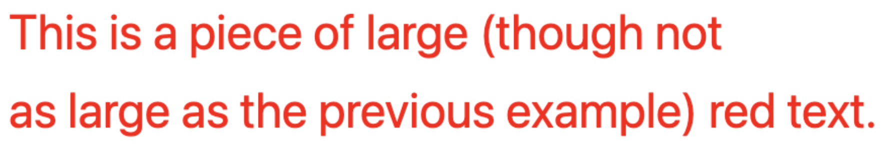
```
<font size=+2>
                <p>This is a piece of large text.</p>
</font>
```


This is a piece of large text.

□

Example 2.2.8 Entering large text with color.

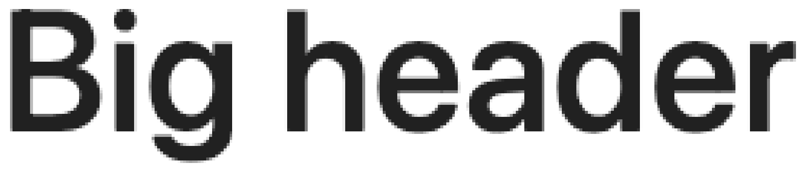
```
<font color='red' size=+1>
                <p>This is a piece of large (though not <br
                    />
                as large as the previous example) red
                    text.</p>
</font>
```


This is a piece of large (though not
as large as the previous example) red text.

The `<br />` command inserts a line break in the text. □

Example 2.2.9 Adding a big header.

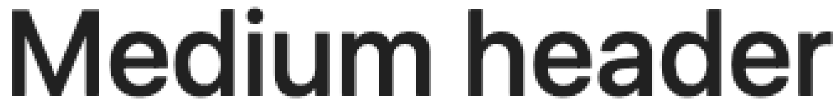
```
# Big header
```


Big header

Equivalently, we could enclose the header text in the HTML style commands `<h1>` and `</h1>`. □

Example 2.2.10 Adding a medium header.

```
## Medium header
```


Medium header

Equivalently, we could enclose the header text in the HTML style commands `<h2>` and `</h2>`. We can get smaller section headers still by putting in another hashtag or two before the text (or using `h3` or `h4` styles). □

2.3 An introduction to SageMath

We will introduce the SageMath computer algebra system and walk you through some commands.

In SageMath, we enter commands and then get them to execute via an "Evaluate" button (if one is present) or by hitting SHIFT-RETURN while the cursor is within the command. For example if we want to factor the integer 1438880, we would give the following command:

```
factor(1438880) #Hit SHIFT-RETURN while your cursor is
   within this cell.
```

We can understand that output easily. In some cases, we may want the result to look more like we would see in a math textbook. In that case, we can wrap our command in a `pretty_print` command.

```
pretty_print(factor(1438880))
```

One of the handiest features built into SageMath is tab completion of commands. Suppose you want to compute 56! but don't remember the exact command to do this. You suspect that the command will have factorial somewhere in its name. To see if that guess is correct, just type the letters `fac` then hit the TAB key. (This works only in SageMath itself, not in the online textbook.)

```
fac  #Put the cursor at the end of the command and hit TAB
```

A pop-up menu tells you that the only two commands that begin with fac are `factor` and `factorial`.

```
factorial(56)
```

```
710998587804863451854045647463724949736497978881168458687447040000000000000
```

SageMath is an *object-oriented* language (it's built on Python), meaning that things like numbers are considered as *objects*, which have methods associated with them. For example, suppose we set $a = 56$, and you were wondering what commands SageMath offers for working with integers like 56. In this case, a is our object and we can find all the methods associated with integers by typing `a.` followed by the TAB key.

```
a = 56 # Hit SHIFT-ENTER
```

```
a. # In a Jupyter notebook, put the cursor after the period
   and hit TAB.
```

You can choose a method from the (long) pulldown menu that results. Methods require parentheses at their end, so even after choosing a method, you need to tack on parentheses. So we can factor a just by entering:

```
a.factor()
```

```
2^3 * 7
```

Some SageMath commands have aliases that look more like function notation. For instance, we can also factor a by typing:

```
factor(a)
```

```
2^3 * 7
```

2.4 SageMath as a calculator

The basic arithmetic operators are `+`, `-`, `*`, and `/`. Sage uses the traditional `^` for exponentiation as well as the Python `**`.

```
print(1+1)
print(103-101)
print(7*9)
print(7337/11)
print(11/4)
print(2^5)
```

Normally, if we have many commands in one cell, SageMath will only print the output from the last command. We have to use the `print` command for Sage to display all the outputs. (Note also that recent versions of Sage are built atop Python 3. Thus we must use the parentheses with the `print` command.)

Sage adheres to the PEMDAS order of operations. When dividing two integers, Sage will return a fractional answer unless we write one of the integers using a decimal.

```
print(11/4)
print(11/4.0)
```

2.5 Solving equations and inequalities

In Sage, equations and inequalities are defined using the operators `==` (equal to), `!=` (not equal to), `<` (less than), `<=` (less than or equal to), `>` (greater than), and `>=` (greater than or equal to) and will return either True or False.

```
9 == 9
```

```
True
```

```
9 == 10
```

```
False
```

To solve an equation or inequality, we use the aptly named `solve()` command. In Sage, x is automatically considered to be a variable. (We will see how to handle other letters shortly.)

```
solve(3*x-2==5,x)
```

```
[x == (7/3)]
```

(Note that you always need the `*` operator on multiplication. No shortening to `3x`.)

```
solve(2*x-5==1,x)
```

```
[x == 3]
```

```
solve(2*x-5>=17,x)
```

```
[[x >= 11]]
```

```
solve(x^2+x==6,x)
```

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

```
pretty_print(solve(e^x == -1,x))
```

If Sage can't solve your equation exactly -- you'll know if Sage simply throws your equation or inequality back at you untouched -- we can use the `find_root` command to get a numerical approximation for the solution. (This command uses Newton's method, among other possible methods.)

```
solve(sin(x)==cos(x),x)
```

```
[sin(x) == cos(x)]
```

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

```
0.7853981633974484
```

2.6 Standard constants and functions

Notice from the last couple of examples that the constants e, i, and π are known to Sage. We can get those constants by typing `e`, `I`, or `pi`, respectively. Typing those will get you exact answers in terms of those constants. If a decimal answer is required, we can use the `n` command to get a numerical approximation.

```
print(e)
print(e.n())
print(e.n(digits=100))
```

```
e
2.71828182845905
2.7182818284590452353602874713526624977572470936999595749669676277240766303535475945713
```

We also have the typical functions: `sin()` and `cos()`, but also `sqrt()`, and `exp()` and all the usual trigonometric and inverse trigonometric functions. One oddity: both `ln()` and `log()` refer to the logarithm to the base e. To get a logarithm to the base b, use the command `log(x,b)`.

```
print(sin(pi/6))
print(cos(pi/4))
print(cos(pi/10))
print(cos(pi/10).n())
print(arctan(1))
print(ln(e))
print(log(e))
print(log(100))
print(log(100.0))
print(log(100,10))
```

```
1/2
1/2*sqrt(2)
1/4*sqrt(2*sqrt(5) + 10)
0.951056516295154
1/4*pi
1
1
2*log(10)
4.60517018598809
2
```

2.7 Declaring Variables

Sage automatically declares the symbol x to be a variable each time it starts. To use other letters (or words), we have to declare them to be variables using the `var()` command. So if we wish to solve a system of linear equations or inequalities in x and y, say, we would type something like the following:

```
var('x␣y')
solve([3*x - y == 2, -2*x -y == 1 ], x,y)
```

```
[[x == (1/5), y == (-7/5)]]
```

```
solve([2*x - y == -1 , 2*x - y == 2],x,y)
```

```
[]
```

```
solve([x-y >=2, x+y <=3], x,y)
```

```
[[x == (5/2), y == (1/2)], [x == -y + 3, y < (1/2)], [x ==
    y + 2, y < (1/2)], [y + 2 < x, x < -y + 3, y < (1/2)]]
```

(In the second case, that system has no solution. In the third, the list of solutions gives the intersection point, then two rays, then the region between the two rays.)

2.8 Calculus

Sage has many commands for the study of limits, derivatives, and integrals. We can *define* our own functions using syntax like the following:

```
f(x) = x*exp(x)
g(x) = x^2*cos(2*x)
h(x) = (x^2+x-2)/(x-4)
```

We can evaluate these functions using intuitive notation:

```
f(1)
```

```
e
```

```
g(2*pi)
```

```
4*pi^2
```

```
h(-1)
```

```
2/5
```

2.8.1 Limits

We can compute $\lim_{x \to 1} f(x)$ by entering the following command:

```
limit(f(x),x=1)
```

```
e
```

We can do the same for $\lim_{x \to 2} g(x)$:

```
limit(g(x),x=2)
```

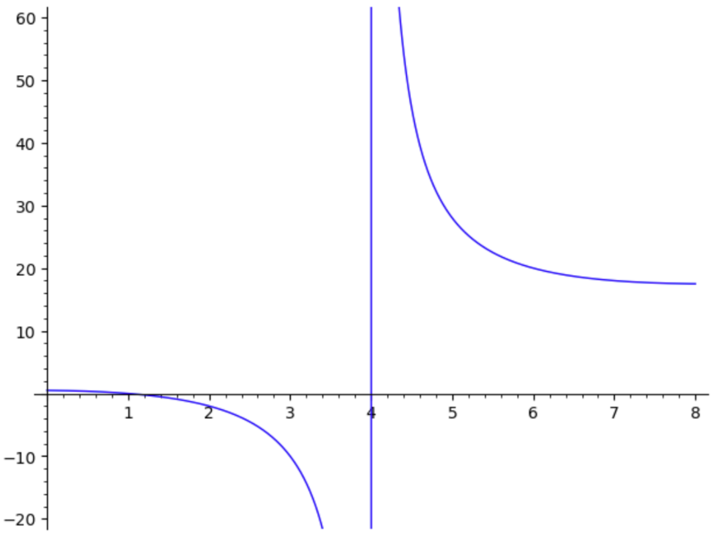
```
4*cos(4)
```

The functions $f(x)$ and $g(x)$ aren't all that exciting as far as limits are concerned since they are both continuous for all real numbers. But $h(x)$ has a discontinuity at $x = 4$.

```
limit(h(x),x=4)
```

```
Infinity
```

This isn't quite right as the following graph shows.



Note that the graph of $h(x)$ has a vertical asymptote at $x = 4$. Thus the overall limit at 4 does not exist. However, we can take the appropriate directional limits as follows:

```
limit(h(x),x=4, dir='right')
```

```
+Infinity
```

```
limit(h(x),x=4, dir='left')
```

```
-Infinity
```

2.8.2 Derivatives

To compute the derivatives of functions, we can use the `diff()` command. (The `derivative()` command works just as well ... but it's longer.)

```
fprime = diff(f,x)
gprime = diff(g,x)
hprime = diff(h,x)
print(fprime(x))
print(gprime(x))
print(hprime(x))
```

```
x*e^x + e^x
-2*x^2*sin(2*x) + 2*x*cos(2*x)
(2*x + 1)/(x - 4) - (x^2 + x - 2)/(x - 4)^2
```

We can now use these functions as any other one.

```
print(fprime(10))
print(gprime(pi/2))
print(hprime(10))
```

```
11*e^10
-pi
1/2
```

For instance, we can now find the critical points of f, g, or h by using the `solve()` command.

```
solve(fprime(x)==0,x)
```

```
[x == -1]
```

```
solve(hprime(x)==0,x)
```

```
[x == -3*sqrt(2) + 4, x == 3*sqrt(2) + 4]
```

2.8.3 Integrals

Sage can compute both definite and indefinite integrals for many common functions.

```
print(integral(f(x),x))
print(integral(g(x),x))
print(integral(h(x),x))
print(integral(h(x),x,0,1))
print(integral(h(x),x,0,1).n())
```

```
(x - 1)*e^x
1/2*x*cos(2*x) + 1/4*(2*x^2 - 1)*sin(2*x)
1/2*x^2 + 5*x + 18*log(x - 4)
18*log(3) - 36*log(2) + 11/2
0.321722695867944
```

Sage loses a point for leaving off the $+C$ on indefinite integrals.

2.9 Plotting

Sage has a basic `plot()` command to produce 2D function graphs. To produce a basic graph of $f(x) = \sin x$ from $x = -\pi/2$ to $x = \pi/2$, we would type

```
plot(sin(x),(x,-pi/2,pi/2))
```

This is rather plain by design. We can add options to make things look a little better.

```
plot(sin(x),(x,-pi/2,pi/2),axes_labels=['$x$','$\\sin\\,x$'],color='purple')
```

We can also change the style of line and its thickness:

```
plot(sin(x),(x,-pi/2,pi/2),linestyle='--',thickness=3)
```

We can graph two functions together on the same set of axes by adding the plots together:

```
p = plot(sin(x),(x,-pi/2,pi/2),color='black')
q = plot(cos(x),(x,-pi/2,pi/2),color='red')
show(p+q)
```

We can now use the `find_root()` command we saw earlier to find the point of intersection of the two graphs and then add this point to the above graph.

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

```
0.7853981633974484
```

```
P = point([0.7853981633974484,sin(0.7853981633974484)]) # I
    used copy and paste to get that number there
T =
    text("(0.79,0.71)",(0.7853981633974484,sin(0.7853981633974484)+0.10))
show(p+q+P+T)
```

Note that Sage handles many of the details of making graphs look "nice." However, sometimes Sage will need our help.

```
f(x) = (x^3 + x^2 + x)/(x^2 - x -2 )
p = plot(f(x),(x,-5,5))
show(p)
```

The vertical asymptotes cause Sage to display rather large y-values, which means that most of the features of the graph are obscured. We can explicitly adjust the vertical and horizontal limits of our plot.

```
show(p,xmin=-2, xmax = 4, ymin = -20, ymax = 20)
```

Chapter 3

Some deeper examples

We just saw how to define our own functions in SageMath (meaning mathematical functions). The nice part is that we can do this very easily, using the symbols we would normally use when we write down a function on paper. We can also plot functions quickly, and solve equations in SageMath involving functions. In this chapter, we will look at some additional examples of these.

3.1 Review of mathematical functions

If we want to define $f(x) = x^3 - x$, then we type

```
f(x) = x^3 -x # this is a typical function
f(2)
```

```
6
```

where we have defined $f(x)$ and asked Sage for the value of $f(2)$. We could instead ask for $f(3)$, or any other value, just by changing the code in the above cell.

If we want to define, say, $g(x) = \sqrt{1 - x^2}$, we would type

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

```
0.968245836551854
```

Note that we got a decimal approximation for the answer because we threw in the decimal point at the end. If you want an exact answer, you can re-execute the cell without that decimal point.

Remember that to evaluate several values, we can use the `print` command. (Without the `print`, Sage would only output the last command.)

```
print(g(-0.1))
print(g(-0.01))
print(g(-0.001))
print(g(-0.0001))
print(g(-0.00001))
print(g(-0.000001))
```

```
0.994987437106620
0.999949998749938
0.999999499999875
0.999999995000000
0.999999999950000
0.999999999999500
```

Since you have taken calculus already, you know that the above is a table to help compute

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

numerically, or help confirm your algebraic answer.

3.2 Review of plotting functions

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

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

Likewise ...

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

The above command is equivalent to `plot(g,0,1)`, and we get the graph on $0 \le x \le 1$. If we leave the interval off the command, Sage aims to give the most likely possible interval.

```
plot(g)
```

Notice that SageMath remembers what f and g are -- we don't have to redefine those functions in every cell. (This is not true in general for SageMathCell, which is one of the reasons we had to remind Sage of f and g's definitions above.) However, if you quit your work in the notebook and come back to it later, you'll have to re-execute the cells with the definitions of f and g before doing any more work with them.

3.3 Miscellaneous stuff on functions

3.3.1 Composition of functions

We can compose two functions very easily in SageMath. Consider the two functions:

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

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, we can print this function, and the values of $h(x)$ whenever we like.

```
f(x) = 3*x + 5
g(x) = x^3 + 1
h(x) = f(g(x))
```

```
print(h(x))
print(h(1))
print(h(2))
print(h(3))
```

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

Recall, though, that generally $g(f(x)) \neq f(g(x))$.

```
h(x) = g(f(x))
print(h(x))
print(h(1))
print(h(2))
print(h(3))
```

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

This is completely different output than what we got previously. We can do the calculations to verify:

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

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

3.3.2 Manipulating polynomials

If we enter the following code

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

```
2*x^2 - 13*x + 21
```

(Don't forget the asterisk for multiplication between the 5 and the x... and the 8 and the x.) Asking for the difference of the two functions gets us the right answer as well:

```
a(x) - b(x)
```

```
3*x-9
```

On the other hand, a request for the product of the two seems unfinished:

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

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

The `expand` method will multiply out that product.

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

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

So g is the function that sends x to $x^4 - 13 * x^3 + 61 * x^2 - 123 * x + 90$.

On the other hand, perhaps we want to factor g instead:

```
g.factor()
```

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

SageMath does allow us to write some methods like `expand` or `factor` in function form instead. For instance, `factor(f)` is equivalent to `f.factor()`. To get rid of the ``evaluates to'' piece, we need to specify the "of x" part of our function.

```
print(factor(g))
factor(g(x))
```

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

```
print(expand(a(b(x))))
factor(a(b(x)))
```

```
x^4 - 16*x^3 + 89*x^2 - 200*x + 156
(x^2 - 8*x + 13)*(x - 2)*(x - 6)
```

```
factor(x^4 - 60*x^3 + 1330*x^2 - 12900*x + 46189)
```

```
(x - 11)*(x - 13)*(x - 17)*(x - 19)
```

Recall that we can find factors of numbers as well. If we were searching for roots of the previous polynomial, we would want to factor the constant term in order to look for possible rational roots.

```
factor(46189)
```

```
11 * 13 * 17 * 19
```

```
divisors(46189)
```

```
[1,
 11,
13,
17,
19,
143,
187,
209,
221,
247,
323,
2431,
2717,
3553,
4199,
46189]
```

Here's a first "program'' that will demonstrate a couple of the commands from above, and the differences in output:

```
f(x) = a(b(x))
print("Direct:")
print(f(x))
print("Expanded:")
print(expand(f(x)))
print("Factored:")
print(factor(f(x)))
```

```
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)
```

3.3.3 Solving problems symbolically

Computer algebra systems like SageMath can solve problems in two ways: symbolically or numerically. When we solve numerically, we get a decimal approximation (a good one!) of the answer. When we solve symbolically, we get an exact answer, often in terms of radicals, logarithms, or other constants and functions. Computer algebra systems like SageMath can solve problems in two ways: symbolically or numerically. When we solve numerically, we get a decimal approximation (a good one!) of the answer. When we solve symbolically, we get an exact answer, often in terms of radicals, logarithms, or other constants and functions.

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

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

Note that this assumes we're looking for the roots of the function we've given. On the other hand, we can give an equation for SageMath to solve:

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

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

By the way, we can use the `pretty_print` command to get nice-looking output:

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

Note the square brackets encompassing the solutions. This is an example of a list in Python.

One way to distinguish symbolic versus numeric solutions is to compare

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

```
1354394/431117
```

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

```
3.14159265350241
```

That last answer is really close to π (relative error: 2.78×10^{-11}).

There is no restriction to answers from polynomials. On the other hand ...

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

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

... only produces one value because `solve` uses the arcsine function to solve the equation. The (infinitely) other values for which $\sin\theta = 1/2$ don't get produced. (The answer is $\theta = \pi/6 + 2\pi \cdot k, 5\pi/6 + 2\pi \cdot k$, for all integers k.)

Notice that we had to declare 'theta' as a variable before solving that equation. Remember that whenever we're using a variable symbolically (as opposed to assigning it a value), we have to declare it in advance. For example, if we want to re-derive the quadratic formula, we could type:

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

There's a similar formula for cubics:

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

SageMath can solve systems of equations, where we use the square brackets to let SageMath know we're providing a list of functions, rather than just one. So to solve the system:

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

we 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)
```

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

In the cases where we will have a lot of possible solutions, it may be good to name the solution list using a variable and then asking for parts of the list:

```
answer = solve( x^6 - 21*x^5 + 175*x^4 - 735*x^3 + 1624*x^2
    - 1764*x + 720 == 0, x)
answer
```

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

Since SageMath is built out of Python, it uses Python's convention of numbering list elements starting from 0 and not from 1.

```
answer[0]
```

```
x == 5
```

```
answer[4]
```

```
x == 3
```

```
answer[-1] # always gives the last element
```

```
x == 1
```

... and a last reminder that the `solve` command is not restricted to polynomial equations.

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

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

As a last piece, there are lots of ways to manipulate function displays.

```
# Logs
f(x) = log(x^2 * sin(x) / sqrt(1 + x))
print("Original␣function:")
f.show()
print("This␣form␣is␣easier␣to␣work␣with:")
show(f.expand_log())
print("Simplify␣expanded␣form:")
show(f.expand_log().simplify_log())
# Rational functions
f(x) = (x + 1) / (x^2 + x)
print("Original␣function:")
f.show()
print("Simplified:")
show(f.simplify_rational())
```

Chapter 4

Basic programming

SageMath is built on Python, which means we can go beyond the one-line computational commands we've learned so far, to write programs with sequences of instructions.

What we present here are the most basic programming constructs; those who are fluent in another programming language (such as C++) will find many familiar structures here.

4.1 Basic Sage/Python syntax

Instructions are generally processed line by line. The hashtag `#` is considered by Python to begin a comment, until the end of that line. A semicolon `;` separates several instructions typed on the same line, although if we're looking to see output, we have to use the `print` command on each.

```
print(2*3); print(3*4); print(4*5)  # one comment, three
    results
```

```
6
12
20
```

In the case where we have a command that's overlong, SageMath will continue type to the right and put a scrollbar in. Another option if we don't wish to scroll is to end a line with a backslash and continue the command on the next line. (The backslash is ignored by SageMath.)

```
123 + \
345
```

```
468
```

4.1.1 Function calls

To evaluate a function, its arguments should be put inside parentheses -- for example, `cos(pi)`. If a function has no argument, we include empty parentheses. If we simply type a function without an argument or parentheses, no computation is performed.

4.1.2 More about variables

As seen previously, SageMath denotes the assignment of a value to a variable by the equals sign. The expression to the right of the equals sign is first evaluated, then its value is saved in the variable whose name is on the left. Thus we have

```
y = 3; y = 3*y + 1; y = 3*y + 1; y
```

```
31
```

The `del x` command discards the value assigned to the variable x, and the function call `reset()` recovers the initial Sage state.

Several variables can be assigned simultaneously, which differs from sequential assignments.

```
a, b = 10, 20 # (a,b) = (10,20) and [10,20] also possible
```

```
a, b = b, a
a, b
```

```
(20, 10)
```

The assignment `a, b = b, a` is equivalent to swapping the values of a and b using an auxiliary variable.

```
temp = a; a = b; b = temp # equivalent to a, b = b, a
(a, b)
```

```
(10, 20)
```

We can also assign the same value to several variables simultaneously.

```
a = b = c = 0
(a, b, c)
```

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

The instructions `x += 5` and `n *= 2` are respectively equivalent to `x = x+5` and `n = n*2`.

The comparison between two objects is performed by the double-equals sign `==`.

```
print(2 + 2 == 2^2)
print(3 * 3 == 3^3)
```

```
True
False
```

4.2 Algorithmics

In many ways, learning to program in SageMath or Python is very similar to learning the basics of English or another spoken or written language. There are very simple instructions at the base -- things like assigning a value to a variable, or getting the result of a computation. There are also compound instructions, like loops and conditionals, that are made up of several instructions, which could be simple or compound themselves.

4.2.1 Loops

An *enumeration loop* performs the same computation for all integer values of some index $k \in \{a, \ldots, b\}$. For example:

```
for k in [1..5]:
        print(7*k)
```

```
7
14
21
28
35
```

The colon at the end of the first line tells SageMath that the block of instructions begins on the next line, and evaluates the product $7k$ for each successive value of k (1, 2, 3, 4, 5). At each iteration, SageMath outputs the product $7k$ via the `print` command.

In this example, the repeated instruction block contains a single instruction, which is indented with respect to the first line. Indenting is extremely important in SageMath and Python, as there are no `begin` ... `end` commands for blocks of instructions. Note that SageMath automatically indented the line after the `for` command.

The following two pieces of code yield different results due to different indenting:

```
S = 0
for k in [1..3]:
        S = S + k
S = 2*S
S
```

```
12
```

```
S = 0
for k in [1..3]:
        S = S + k
        S = 2*S
S
```

```
22
```

In the first block, the instruction `S = 2*S` is executed only once at the end of the loop, while in the second it is executed at every iteration, which explains the different results:

$$S = (0 + 1 + 2 + 3) \cdot 2 = 12 \qquad S = ((((0 + 1) \cdot 2) + 2) \cdot 2) + 3) \cdot 2 = 22.$$

(Note that we can stop typing code in as part of a loop simply by hitting backspace at the start of a line.)

This kind of loop will be useful to compute a given term of a recurrence. The syntax is very simple and can be used without any problem for thousands, tens of thousands, or hundreds of thousands of iterations. The main problem is that it constructs the list of all possible values of the loop variable first before executing the instructions in the block. Some of the options below can make the repetitions go faster.

Table 4.2.1

Iteration functions of the `..range` for `a`, `b`, `c` integers	
`for k in [a..b]: ...`	constructs the list of SageMath integers $a \leq k \leq b$
`for k in srange (a, b): ...`	constructs the list of SageMath integers $a \leq k < b$
`for k in range (a, b): ...`	constructs a list of Python integers (`int`)
`for k in xrange (a, b): ...`	enumerates Python integers (`int`) without explicitly constructing the corresponding list
`for k in sxrange (a, b): ...`	enumerates SageMath integers without constructing a list
`[a,a+c..b], [a..b, step=c]`	SageMath integers a, $a+c$, $a+2c$, ... as long as $a+kc \leq b$
`..range(b)`	equivalent to `.. range (0, b)`
`..range(a, b, c)`	sets the iteration increment to c instead of 1

4.2.2 While loops

The other kind of loops are `while` loops. The difference here is that, as opposed to enumeration loops, rather than execute a set of instructions a fixed number of times, SageMath will execute the instructions based on a given condition being true.

```
S = 0; k = 0   # The sum S is initialized to 0
while e^k <= 10^6:  # e^13 <= 10^6 < e^14
        S = S + k^2  # accumulates the squares k^2
        k = k + 1
S
```

```
819
```

A connection you may have noticed if you've taken Calculus II (or soon will notice) is the connection between loops and mathematical constructs like sums and sequences. In fact what we just computed was

$$S = \sum_{\substack{k\in\mathbb{N}\\ e^k \leq 10^6}} k^2 = \sum_{k=1}^{13} k^2 = 819, \qquad e^{13} = 442413 \leq 10^6 < e^{14} = 1202604$$

Notice that there are two instructions inside the loop block: one to accumulate the new square, and the next to move the index k to the next value.

4.2.3 Exercises

1. Use a `while` loop to solve the following problem: With $x = 10000$, determine the unique integer n such that $2^n \leq x < 2^{n+1}$.
2. Compute the sum of the first 1001 positive integers. Print only the final total.
3. Use a loop to investigate $\lim_{x\to 8} e^{1/(8-x)}$. In other words, find a way to produce values for $f(x) = e^{1/(8-x)}$ as $x \to 8$. (Don't forget we're checking a two-sided limit. A plot may be helpful.)

4.3 More Algorithmics

When we use loops to repeat instructions, we can have a loop that carries on longer than we wish. For instance, we may be looking for the first instance of an inequality to be true. We find it, but the loop carries on for several more iterations anyway. With `while` loops, we can accidentally write code so that the condition for the loop to terminate never becomes true, and computation goes on and on and on.

4.3.1 Aborting a loop execution

The `for` and `while` loops repeat a given number of times the same instructions. The `break` command inside a loop interrupts it before its end, and the `continue` command goes directly to the next iteration. Those commands allow to check the terminating condition at every place in the loop.

The four examples below determine the smallest determine the smallest positive integer x satisfying $\log(x+1) \leq x/10$.

```
for x in [1.0..100.0]:
        if log(x+1) <= x/10:
                break
x
```

```
37.0000000000000
```

```
x = 1.0
while log(x+1) > x/10:
                x = x + 1
x
```

```
37.0000000000000
```

```
while log(x+1) > x/10 and x < 100:
                x = x + 1
x
```

```
37.0000000000000
```

```
x = 1.0
while True:
        if log(x+1) > x/10:
                x = x+1
                continue
        break
x
```

```
37.0000000000000
```

The first of the four examples uses a `for` loop with at most 100 tries which terminates terminates once the first solution is found; the second program looks for the smallest solution and might not terminate if the condition is never fulfilled; the third is equivalent to the first one with a more complex loop condition; finally the fourth has an unnecessarily complex structure, whose unique goal is to exhibit the `continue` command. In all cases the final value x is 37.0.

4.3.2 Conditionals

Notice that we had a new keyword -- `if` -- in the first example. The last two lines are an example of a *conditional*. This is a test that enables us to execute code based on the result of a boolean (true/false) condition. Some possible examples of the syntax are:

```
if a condition:
        an instruction sequence
```

```
if a condition:
        an instruction sequence
else:
        another instruction sequence
```

As an example, there is the famous Collatz sequence defined by:

$$x_0 \in \mathbb{N} \setminus \{0\}, \qquad x_{n+1} = \begin{cases} x_n/2 & \text{if } x_n \text{ is even} \\ 3x_n + 1 & \text{if } x_n \text{ is odd} \end{cases}$$

The Collatz conjecture (which is unsolved) says that for all possible starting values x_0, the sequence will eventually reach a 1, after which it will cycle 4, 2, 1, 4, 2, 1 ... forever.

```
x = 6; n = 0
while x != 1:    # the test x <> 1 is equivalent
    if x%2 == 0: # the operator % yields the remainder
        x = x//2 # //: division quotient
    else:
        x = 3*x + 1
    n = n + 1
    print(n,x)
```

We check whether x_n is even by seeing whether the remainder of the division of x_n by 2 is 0. The variable n at the end of the block is the number of iterations. The loop ends as soon as $x_n = 1$. So this result says that if $x_0 = 6$ then $x_8 = 1$, and $8 = \min\{p \in \mathbb{N} \mid x_p = 1\}$.

The `if` instruction also allows nested tests in the `else` branch using the `elif` (short for *else if*) keyword. These two following structures would then be equivalent:

```
if a condition cond1:
        an instruction sequence inst1
else:
        if a condition cond2:
                an instruction sequence inst2
        else:
                if a condition cond3:
                        an instruction sequence inst3
                else:
                        in all other cases inst4
```

```
if cond1:
        inst1
elif cond2:
        inst2
elif cond3:
        inst3
else:
        inst4
```

4.3.3 Procedures and Functions

As in other computer languages, we can define our own procedures and functions, using the `def` command whose syntax we detail below. (The difference between a function and a procedure is that a function always returns a result, whereas a procedure does not.) Either can be defined to take as input one or several arguments, or none at all. For instance, we can define the function that takes x and y to $x^2 + y^2$ in this way:

```
var('x␣y')
fct1(x,y) = x^2 + y^2
```

```
var('a')
fct1(a,2*a)
```

```
5*a^2
```

By default, all variables appearing in a function are local variables. That is, they are created at each call to the function, used as needed within the function, and destroyed at the end of the function, and do not interact with other variables of the same name. In particular, global variables (variables used outside the function) are not modified.

```
def foo(u):
        t = u^2
        return t*(t+1)
```

```
t = 1; u = 2
foo(3), t, u
```

```
(90, 1, 2)
```

It *is* possible to modify a global variable from within a function, using the `global` keyword.

```
a = b = 1
def f():
    global a
    a = b = 2
f(); a, b
```

```
(2, 1)
```

Example 4.3.1 Working with Coins. We're going to start with a very easy subroutine, which would be part of the programming of a vending machine or a cash register. It is going to reply with the dollar value for some number of quarters, dimes, nickels, and pennies, which of course have respective values \$0.25, \$0.10, \$0.05, and \$0.01. This is a simple task, so that we can concentrate on the programming rather than the underlying mathematics.

```
def coinCount( quarters,dimes,nickels,pennies ):
   """
␣␣␣This␣subroutine␣takes␣in␣data␣about␣a␣pile␣of␣change,␣
    with␣the␣four␣parameters␣being
␣␣␣the␣number␣of␣quarters,␣dimes,␣nickels,␣and␣pennies.␣␣
    Then␣the␣total␣dollar␣value␣of
␣␣␣these␣is␣reported.␣␣The␣quantities␣are␣assumed␣to␣be␣
    natural␣numbers.␣␣Please␣do
␣␣␣not␣use␣complex␣numbers␣as␣inputs,␣as␣it␣will␣horrify␣a␣
    user␣to␣find␣out␣that␣some␣of
␣␣␣their␣money␣is␣imaginary.
␣␣␣"""
   total = quarters*0.25 + dimes*0.10 + nickels*0.05 +
       pennies*0.01
   print("That's␣a␣total␣of␣$"),
   print(total)
```

The first line, with the `def` command, signals Python and Sage that we're about to define a new subroutine. We follow this immediately with the name of the subroutine, and then a list of the variables needed in that subroutine. Here, we have variables for quarters, dimes, nickels, and pennies. While in algebra, variables tend to be a single letter -- in programming, we use words so that we don't have to remember what each variable stands for. The colon is very important. It indicates that the next few commands are subordinate to the `def` statement. The indentation shows specifically which statements are subordinate.

After that, on the second line, we have a formula that computes the total value of the coins. Each coin is multiplied by its value in dollars, and the total is placed in the variable which is unsurprisingly called "total." The third line and the fourth line print the amounts in a nicely formatted human-readable way. (Note that there's a description of the subroutine at the top, set off by triple quotation marks.)

We can test our subroutine with 5 quarters, 4 dimes, 3 nickels, and 2 pennies.

```
coinCount(5,4,3,2)
```

```
That's␣a␣total␣of␣$
1.82000000000000
```

It is important to remember that order matters. Consider the following three calls to this function.

```
coinCount(1, 3, 2, 1)
```

```
That's␣a␣total␣of␣$
0.660000000000000
```

```
coinCount(1, 1, 2, 3)
```

```
That's␣a␣total␣of␣$
0.480000000000000
```

```
coinCount(1, 2, 3, 1)
```

```
That's␣a␣total␣of␣$
0.610000000000000
```

As you can see, all three calls are asking about different piles of change, and those piles will have different total values. You could imagine that someone using the subroutine on three different days, who has forgotten the ordering, might guess the ordering and input those three lines above. The output produces below illustrates the point: you get a wrong answer if you put the parameters in the wrong order.

Unlike most computer languages, Python has a solution for those who cannot remember orderings well. Using the following notation, we label each value with the variable that we hope it will be assigned to. We will get the same answer each time.

```
coinCount( dimes=3, nickels=2, pennies=1, quarters=1 )
coinCount( quarters=1, pennies=1, nickels=2, dimes=3 )
coinCount( pennies=1, nickels=2, dimes=3, quarters=1 )
```

```
That's␣a␣total␣of␣$
0.660000000000000
That's a total of $
0.660000000000000
That's␣a␣total␣of␣$
0.660000000000000
```

This is a very nice service! The one thing that Python and Sage will always ding us for are typos or misspellings of variables.

```
coinCount( dimes=3, nickels=2, pennies=1, quaters=1 )
```

```
---------------------------------------------------------------------------
TypeError                                     Traceback (most
    recent call last)
Cell In[1], line 1
----> 1 coinCount( dimes=Integer(3), nickels=Integer(2),
    pennies=Integer(1), quaters=Integer(1) )

TypeError: coinCount() got an unexpected keyword argument
    'quaters'
```

□

Checkpoint 4.3.2 Simulate a cash register as follows. Suppose a small foodstand in a subway station sells sodas ($1.50 each), bananas ($0.75 each), sandwiches of various types (all $3.75 each), and bottles of water ($2 each). Write a subroutine that takes five inputs: the number of sodas, bananas, sandwiches and bottles of water, as well as the amount of cash tendered (for example, $20 or $10). Your program should tell the user how much change, in dollars and cents, is therefore due.

Checkpoint 4.3.3 Designing Aquariums. Now we're going to write a subroutine that is suitable for a company that builds custom display aquariums for use in dentist's offices and other upscale businesses. Customers come with orders and the cost must be correctly calculated, based on the sizes of the six panels which comprise the rectangular aquarium.

It turns out that the particular materials that the company is using have the following costs:

- The bottom of the aquarium is metal, and costs 1.3 cents/sq in.
- The top of the aquarium is plastic, and costs half a cent/sq in.
- The sides of the aquarium are glass, and cost 5 cents/sq in.

Given a customer preference for the volume of the aquarium, there are many choices of dimensions and they will have substantially different costs. Consider a customer who wants a 12,000 cubic inch aquarium. (Note that 12,000 cubic inches is about 52 gallons so that is not a very large aquarium at all.) We can compute the following prices:

- A choice of $20 \times 30 \times 20$ has a cost of \$110.80.
- A choice of $40 \times 30 \times 10$ has a cost of \$91.60.
- A choice of $60 \times 20 \times 10$ has a cost of \$101.60.
- A choice of $80 \times 15 \times 10$ has a cost of \$116.60.

Those prices were computed using the `analyzeAquariumDesign` subroutine, which is found below. The commands calling the subroutine are found below the code.

```
top_cost = 0.5
bottom_cost = 1.3
side_cost = 5

def analyzeAquariumDesign( length, width, height):
    """
␣␣␣␣Here␣the␣dimensions␣of␣an␣aquarium␣are␣the␣inputs.␣The
␣␣␣␣sizes␣of␣the␣six␣panels␣that␣form␣the␣rectangular␣
    aquarium
␣␣␣␣will␣be␣printed␣on␣the␣screen,␣along␣with␣their␣costs.␣
    The
␣␣␣␣costs␣are␣computed␣in␣terms␣of␣the␣areas␣of␣those␣panels␣
    and
␣␣␣␣the␣global␣variables␣top_cost,␣bottom_cost,␣and␣
    side_cost.
␣␣␣␣Finally,␣the␣volume␣and␣the␣total␣cost␣are␣displayed.
␣␣␣␣"""
    bottom_top_area = length*width
    front_back_area = length*height
    left_right_area = width*height

    print ("The␣left/right␣panels␣have␣an␣area␣of",
        left_right_area,
           "costing␣$",
               round(side_cost*left_right_area*0.01,2))
    print ("The␣front/back␣panels␣have␣an␣area␣of",
        front_back_area,
           "costing␣$",
               round(side_cost*front_back_area*0.01,2))
    print ("The␣top␣panel␣has␣an␣area␣of", bottom_top_area,
           "costing␣$",
               round(top_cost*bottom_top_area*0.01,2))
    print ("The␣bottom␣panel␣has␣an␣area␣of",
        bottom_top_area,
           "costing␣$",
               round(bottom_cost*bottom_top_area*0.01,2),"\n")

    print ("The␣total␣volume␣is", N(
        length*width*height,digits=2 ),"\n")

    cost = top_cost*bottom_top_area +
        bottom_cost*bottom_top_area + \
    2*front_back_area*side_cost + 2*left_right_area*side_cost

    print ("The␣total␣cost␣is␣$",
        round(cost*0.01,ndigits=2),"\n")
```

```
analyzeAquariumDesign( 20, 30, 20 )
analyzeAquariumDesign( 40, 30, 10 )
analyzeAquariumDesign( 60, 20, 10 )
analyzeAquariumDesign( 80, 15, 10 )
```

```
The left/right panels have an area of 600 costing $ 30.0
The front/back panels have an area of 400 costing $ 20.0
The top panel has an area of 600 costing $ 3.0
The bottom panel has an area of 600 costing $ 7.8

The total volume is 12000.

The total cost is $ 110.8

The left/right panels have an area of 300 costing $ 15.0
The front/back panels have an area of 400 costing $ 20.0
The top panel has an area of 1200 costing $ 6.0
The bottom panel has an area of 1200 costing $ 15.6

The total volume is 12000.

The total cost is $ 91.6

The left/right panels have an area of 200 costing $ 10.0
The front/back panels have an area of 600 costing $ 30.0
The top panel has an area of 1200 costing $ 6.0
The bottom panel has an area of 1200 costing $ 15.6

The total volume is 12000.

The total cost is $ 101.6

The left/right panels have an area of 150 costing $ 7.5
The front/back panels have an area of 800 costing $ 40.0
The top panel has an area of 1200 costing $ 6.0
The bottom panel has an area of 1200 costing $ 15.6

The total volume is 12000.

The total cost is $ 116.6
```

Part II

Mathematical Modeling

Chapter 5

Introduction to Mathematical Modeling

Every student of mathematics has done some "mathematical modeling" in his/her educational career. These instances of mathematical modeling are typically called "applications" and are used to illustrate how mathematics is implemented in the "real world."

In most math classes, the main goal is to learn the theory of some particular mathematical discipline. The applications are used to help achieve this goal by providing a more concrete context in which to study and understand the theory. For instance, in Calculus I, the real goal is to understand the idea of the limit and the derivative. An applied maximization problem is used to motivate the idea of the derivative and to provide practice in calculating and interpreting derivatives.

In mathematical modeling, the opposite is true. Here we will start with some "real world" problem and use mathematical theory and techniques to better understand the phenomena behind the problem.

5.1 Definitions

To define the phrase *mathematical modeling*, we will first define the term model. The word model is used frequently in everyday language.We talk about model airplanes, model houses, models on a runway, etc. What does the term model mean in a mathematical sense?

Lucas (Lucas, William F., The Impact and Benefits of Mathematical Modeling, in *Applied Mathematical Modeling* (D.R Shier and K.T. Wallenius eds.), Chapman and Hall/CRC, 1999, pg. 5) defines a model as "a simpler realization or idealization of some more complex reality." The real world is a very complex place. To better understand it, we need to try to simplify it to a reasonable degree, describe the simplification in ways we can understand and work with, and then study the simplification. This is what we call *modeling.*

A *mathematical model* then can be defined as a model constructed using mathematical terms, symbols, and ideas. Giordano et. al. (Frank R. Giordano, M. D. Weir, and W. P. Fox, *A First Course in Mathematical Modeling*, Third ed., Thomson Brooks/Cole, 2003, pg. 54) defines a mathematical model as "a mathematical construct designed to study a particular real world system or phenomenon." Mathematical models can take many different forms. They may involve equations, inequalities, differential equations, matrices, logic, or any other type of "mathematical" idea.

The key idea is that we use mathematics to describe a portion of the real world. Therefore, a very simple but general definition of the process of mathematical modeling is:

Definition 5.1.1 Mathematical modeling is the application of mathematics to real world problems. ◊

5.2 Purpose

Why do we do mathematical modeling? Since we want to answer a question about real world phenomena, we could just sit back, observe, and take notes. Suppose we put 500 bacteria in a Petri dish. The next day we count 525 in the dish, and the next we count 551.

Obviously, the number of bacteria is growing. Based on this observation, we might ask these questions:

- How long will it be until there are 600 bacteria in the dish?
- If we need 900 bacteria for an experiment in 3 days, how many must we put into the dish today?

We could answer each question as follows:

- Wait until we count 600 bacteria in the dish.
- Put 1 bacterium in a dish, 2 in a second dish, 3 in a third, etc. up to 900, wait 3 days, and determine which dish contains 900 bacteria.

These solutions only require us to make simple observations of this real world phenomenon of bacteria growing in a Petri dish. However, these solutions are obviously impractical for they might require too much time or too many resources (the second solution requires 900 Petri dishes and a total of $1 + 2 + \cdots + 900 = 405,450$ bacteria).

A much more practical approach to answering these questions is to construct a function that gives the number of bacteria in the dish in terms of time (i.e. construct a mathematical model of the bacteria growth).

In other situations, making observations may itself be a complicated ordeal. For instance, suppose we wanted to find the optimal mixture of doctors and nurses (i.e. the number of doctors and number of nurses) to staff a hospital emergency room. The concept of "optimal" may take into account several factors, including:

- Quality of patient care. (Do they get the care they need?)
- Patient waiting time. (Do they have to wait a long time?)
- Time spent with patients. (Are the doctors and nurses over-worked, or do they have too much "free time?")
- Resources. (Is there enough floor space or are people running into each other?)

One approach to finding an optimal number is to choose some mixture of doctors and nurses (say 3 doctors and 8 nurses), put them to work, and have a team of people record data for a series of weeks or months. Then choose another mixture (say 2 doctors and 7 nurses) and repeat the process. Repeat this until all possible combinations of doctors and nurses have been tried, analyze the data, and pick the optimal mixture.

This approach has many of the same problems as the bacteria growth problem. It would take too much time and be too expensive. Plus there are additional problems. If there are too few doctors and nurses on staff, patients might unnecessarily die. Plus we may not observe how the different mixtures handle infrequent events such as a bus crash that floods the ER with dozens of patients at once.

A much more practical solution would be to try to replicate the behavior of the ER on a computer (i.e. create a type of mathematical model called a simulation) where the numbers of doctors and nurses can be easily changed. Each mixture can be simulated for a long period of time under many different situations and at low cost. Plus, nobody dies.

5.3 Process

We will illustrate the process of mathematical modeling with an example of modeling the number of bacteria in a Petri dish as described in Section 5.2.

Step 1: State the question to be answered.

In many situations, this step is almost trivial; in others it is the most difficult part of the process. The question should be narrow enough to make the problem manageable, but not too narrow so that the problem is trivial. Initially we may want to focus on a narrow question, and then use the knowledge gained to broaden the question at a later time. The question should also be stated in precise mathematical terms so it can easily be translated into mathematical notation.

In this example we will answer the question "How long will it be until the number of bacteria in the dish reaches 600?"

Step 2: Select the modeling approach.

In this step we determine the form of the model. In some situations this is easy to do; in others we may have several reasonable choices. Making the right choice requires at least some knowledge of all the possibilities. It also depends on the nature of the assumptions being made.

Often times this step begins with some simple observations. Note that we started with 500 bacteria. After 1 day, it increased by 25, which is 5% of 500. After a second day it increased by 26, which is approximately 5% of 525. The growth rate (or change per day) appears to be relatively constant. This suggests a simple relationship between the populations on consecutive days:

$$\text{Population on one day} = \text{Population on previous day } + 5\%$$

This relationship indicates that we may be able to derive a simple equation to model the population.

Step 3: Define variables and parameters.

Variables are quantities that could change within a problem. *Parameters* are quantities that are constant within a problem, but that could change between problems of the same type. The first part on this step is to determine what variables and parameters are involved. This may be simple and obvious, or very complicated. Often times there are potentially hundreds of quantities involved. To make the model manageable, we need to make assumptions as to which are the most important and which can be ignored. At a later time we could add additional variables and parameters to refine the model.

In this example, variables include:

1. Time
2. Population
3. Temperature
4. Amount of food present
5. Amount of available space in the dish

These are all values that change as the population grows. Since the initial observation did not give any information on temperature, food, or space, we will ignore these variables and focus on only time and population.

Possible parameters to consider include:

1. The initial population
2. Growth rate (we will assume this is constant)
3. Size of the dish
4. Initial amount of food

These are all values that are constant once we put the bacteria in the dish and allow them to grow. But if we consider a different dish with a different population of bacteria, they could change. Again, since we don't know anything about the size of the dish or the amount of food, we will ignore these parameters.

The second part of this step is to choose symbols to represent the variables and parameters. For this example, let

$$\begin{aligned} n &= \text{time in days from the present } (n = 0, 1, \dots) \\ r &= \text{the growth rate (in decimal form)} \\ a_n &= \text{the population at the beginning of day } n \\ a_0 &= \text{the initial population} \end{aligned}$$

Step 4: State the assumptions.

Making assumptions is an essential aspect of creating a valid and manageable model. Assumptions fall into many different categories. Some are used to simplify the model, such as those used to select the important variables. Some are needed to define relationships between the variables because the precise relationships are not known. Others are needed to determine the values of parameters when the exact values are not known.

Clearly stating the assumptions is an important part of interpreting and presenting the results. The results of a model are only as valid as the underlying assumptions. If the assumptions are unreasonable, then the conclusion will be unreasonable regardless of the precision of the mathematical analysis.

In this problem, we have already chosen to ignore temperature, size of the dish, and many other possible variables and parameters. This is a simplification. Furthermore, we will assume that the population growth is constant (*i.e.* the population will increase 5% each day).

Step 5: Formulate the model.

This is where the "mathematics" starts. We have observed that the number of bacteria on day 1 is equal to the number on day 0 plus 5%. The number on day 2 is equal to the number on day 1 plus 5%, etc. In mathematical notation using our variables and parameters, we have

$$\begin{aligned} a_1 &= a_0 + r\,a_0 = (1+r)a_0 \\ a_2 &= a_1 + r\,a_1 = (1+r)a_1 \\ &\vdots \\ a_{n+1} &= a_n + r\,a_n = (1+r)a_n \end{aligned}$$

This forms a recursively defined sequence. To form an explicit description of a_n in terms of n, note that

$$\begin{aligned} a_1 &= (1+r)a_0 \\ a_2 &= (1+r)a_1 = (1+r)(1+r)a_0 = (1+r)^2a_0 \\ a_3 &= (1+r)a_2 = (1+r)(1+r)^2a_0 = (1+r)^3a_0 \\ &\vdots \\ a_n &= (1+r)^n a_0 \end{aligned}$$

This last equation is our model.

Step 6: Solve the model and state the solution.

Here we use the term "solve" loosely. Solving a model may involve solving a single equation, as in this example, it may involve constructing a graph and qualitatively describing its behavior, or it may involve running a simulation several hundred times and summarizing the resulting data. The meaning of the term solve is relative to the type of model.

In this example, the question is "when will the population be 600?" In terms of our variables, this can be stated as "find n such that $a_n = 600$." This yields the equation

$$600 = (1+0.05)^n \cdot 500$$

Solving this equation using logarithms yields $n \approx 3.7$. This means that at the beginning of the fourth day we will have over 600 bacteria. This is our solution.

Often times the results of a model are used to guide decisions. In many practical situations, such as in business or the military, the person doing the modeling is not the final decision maker. The final decision maker is a CEO or officer who is not a mathematician. Therefore, the solution should be stated in as non-technical language as reasonably possible.

Step 7: Verify the model.

Verification is necessary to test the reasonableness of our assumptions. Typically we verify a model by comparing it to some real world data. Let's suppose we let the bacteria grow for a total of 7 days and collect the data in Table 5.3.1. Next to the actual observed populations are the populations predicted by the model.

We see that on day 4, the actual population is just below 600. Even though the predicted population does not equal the actual population, our solution of 4 days is reasonable.

Note that on days 5 and 6, the actual and predicted populations differ considerably. The data indicates that the growth rate slows down. This means that our assumption of a constant growth rate is incorrect.

Table 5.3.1

Day	**Actual Population**	**Predicted Population**
0	500	500
1	525	525
2	551	551
3	575	579
4	598	608
5	610	638
6	620	670

Our model is accurate up to day 4, but inaccurate for later days. This example illustrates that we must be very cautious about using data from the past to make predictions about the future.

Step 8: Refine the model.

Refine means to improve the model in some way. One way to do this is to add variables that we chose to ignore in step 3 to make a more accurate model. Another way is to generalize the model so that it can be used to solve other similar problems. Either one will require us to repeat steps 3 – 7 to some degree.

We have already noted that the data indicates the growth rate slows down over time. This could be a result of diminishing food supplies or room to grow. These are two variables we chose to ignore.

One possible refinement is to redo the model incorporating these two variables. This would require additional observations and data to determine how these variables are related to the other variables. Another possible refinement is to use the available data to model a decreasing growth rate. We will illustrate how to do this in the next chapter.

A simple diagram that illustrates the basic process of mathematical modeling is given in Figure 5.3.2. The process begins in the upper left-hand corner with observations (or data). From this we get the basic problem we want to solve. We then make assumptions, construct the model, solve it using appropriate mathematical tools, and obtain a mathematical result. Then we must interpret the mathematical result in light of the assumptions to make our conclusions. We then verify the model using more observations.

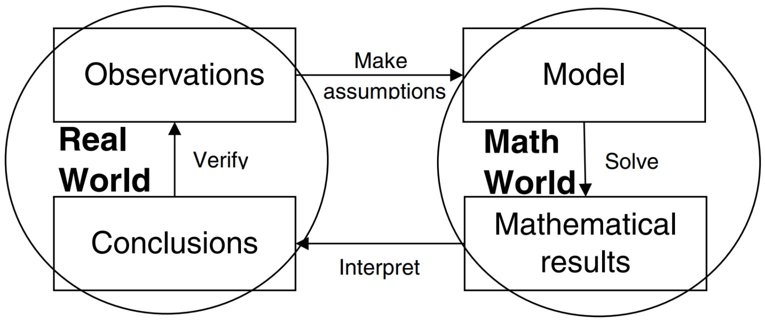


Figure 5.3.2

This figure also illustrates the cyclic nature of mathematical modeling. We rarely stop once we answer the original question. We continually repeat the process, to some extent, to test, refine, and implement the model.

The right half of this diagram is done in the "math world" and the left half is done in the "real world." In the math world, we use the absolute certainty of mathematics. The real world contains no such certainties. Making assumptions and interpreting are necessary steps to move between these two worlds.

5.4 Assumptions

Every model is based on some set of assumptions. Sometimes those assumptions are rather trivial and obvious, but most times they are significant enough to potentially affect the validity of the model. We will never be able to describe each component of a real world system exactly. Assumptions are needed to fill in these gaps, and whenever possible, the reasonableness of assumptions should be tested. In fact, assumptions are so important that any mention of a model should include the assumptions behind it.

Here are a few examples of well–known models and some of their underlying assumptions.

Example 5.4.1 Range of an Electric Car. When considering the purchase of an electric car, the first question most consumers ask is, "what's the range?" The exact answer depends on many factors including the battery size and condition, driving style, vehicle speed, wind speed and direction, temperature, and elevation change. Any numeric answer to this question must be accompanied with many assumptions about these factors. □

Example 5.4.2 Carbon-14 Dating. Carbon–14 dating methods are used to date organic material, such as a piece of bone, found at archaeological sites. The underlying mathematical model requires knowledge of the proportion of Carbon–14 originally present in the sample. Obviously we cannot measure this precisely, so we assume that this proportion is the same as in a modern bone. This assumption can't be tested directly, but if a date can be confirmed via independent means, it would be an indication that the assumption is correct. □

Example 5.4.3 Newtonian Mechanics. Newton's second law says that the force exerted on an object is equal to its rest mass times the acceleration, or $F = ma$. This was assumed to hold at any velocity. In the 20th century, it was discovered that for speeds approaching the speed of light, this rest mass must be replaced with the relativistic mass, which is larger. □

Example 5.4.4 Relativity. Einstein's theory of special relativity is really a mathematical model. It is based on several postulates (a type of assumption), one of which is that the speed of light in a vacuum is constant to any observer in an inertial frame of reference. These assumptions cannot be tested in every circumstance imaginable, but the results of Einstein's theory can be tested. These results have been shown to be correct, so it suggests that the underlying assumptions are also correct. □

Exercises

1. Each part below describes a calculated number. Think about how this number was calculated and identify at least one assumption behind the calculation.

- **(a)** A consumer magazine reports that a laundry detergent costs \$0.31 per load.
- **(b)** An exercise bike displays the number of calories burned.
- **(c)** A jogging pedometer measures the distance jogged.
- **(d)** A dashboard display in a car shows that it can travel 220 miles on the remaining fuel in the tank.
- **(e)** A small business owner predicts that his company will spend \$500,000 on phone bills next year.
- **(f)** A cooking magazine lists one ingredient for a recipe as 1-2 cups of shredded cheddar cheese. It then claims that each serving has 320 calories.

2. Identify at least two possible variables and two parameters involved in each of the following models. Clearly identify which is a variable and which is a parameter.

- **(a)** A biologist wants to model the population of foxes and rabbits in a forest over a period of time.
- **(b)** A consumer wants to model the monthly balance in his credit card.
- **(c)** A public health researcher wants to model the amount of alcohol in the bloodstream of a college student during an evening of partying.
- **(d)** An ecologist wants to model the amount of pollution in a lake over a period of time.
- **(e)** A market researcher wants to model the number of viewers of a TV show over the span of the season.

3. To predict the time at which the bacteria population in Section 5.3 reached 600, we solved the equation $600 = (1 + 0.05)^n \cdot 500$ and concluded that at the beginning of day 4 we will have over 600.

- **(a)** Show how this equation was solved for n.
- **(b)** A student argues that we should be more precise in our conclusion. She argues that we should say, "on day 3.736850652 there will be exactly 600 bacteria" because this is the value given by her calculator. How would you explain to her that such a level of precision is not appropriate?

4. A cashier at a supermarket can checkout, on average, 3 customers a minute (this is called the *service rate*). Customers arrive, on average, 2 a minute (this is called the *arrival rate*). The manager figures that since the service rate is greater than the arrival rate, customers will never have to wait in line. What assumptions is the manager making? Do these assumptions seem reasonable? What does this say about the validity of the conclusion?

5. A college student plans to ask 100 different girls for a date. He calculates the number who will say yes with the following reasoning:

> Since every girl can say yes or no, exactly half will say yes.
> Since half of 100 is 50, exactly 50 girls will say yes.

What, if anything, is faulty with this model? Explain.

6. A model for the population of the United States (in millions) for the years 1800-1960 is

$$P(t) = 0.0063t^2 + 0.0719t + 5.0747$$

where t is the number of years since 1800. A student predicts that the population in the year 2050 will be $P(250) \approx 416.8$ million. What assumption is the student making when doing this calculation? Do you think this assumption is reasonable? What does this say about the validity of the prediction?

Chapter 6

Proportionality and Geometric Similarity

A *model* is “a simpler realization or idealization of some more complex reality.” The real world is a very complex place. To better understand it, we need to try to simplify it to a reasonable degree, describe the simplification in ways we can understand and work with, and then study the simplification. This is what we call *modeling.* A *mathematical model* is a model constructed using mathematical terms, symbols, and ideas. Others have described it as "a mathematical construct designed to study a particular real world system or phenomenon."

6.1 Scatter Plots and Lines of Best Fit in SageMath

One of the first steps in modeling is to make simplifying assumptions, and one of the most basic types of assumptions is that one variable is simply a constant multiple of the other. This type of relationship is called *proportionality.* We say that y is *proportional* to x if there is a nonzero constant c such that $y = cx$. Note that if $y = cx$, then $x = \frac{1}{c}y$ so that x is proportional to y. That is, x is proportional to y if and only if y is proportional to x. (The proportionality relationship is *symmetric.*) So we simply say x and y are proportional.

This means that the graph of y versus x should form a straight line through the origin.

One famous proportionality relationship is *Hooke’s law*, which relates the force applied to a spring to the distance it is stretched or compressed. Hooke’s law simply states that $d = kF$ where F is the force applied to a spring, d is the distance stretched or compressed, and k is a constant related to the stiffness of the spring.

Here is an example. Suppose that we hang a bucket from a spring, fill the bucket with varying amounts of sand, and measure the distance the spring is stretched. The results are recorded in Table 6.1.1.

Table 6.1.1

Weight (newtons)	Distance (cm)
5	1.02
10	1.86
15	3.00
20	3.94
25	4.95
30	5.82
35	6.95
40	7.80

We plot this data to (1) verify that Hooke's law holds for this spring and (2) find the constant of proportionality (the *spring constant*). This data comes from the real world, so it is subject to errors and uncertainty. Therefore, we cannot expect the data to lie perfectly on a straight line as predicted by the idealized Hooke's law. However, the data should lie *very near* a straight line. The slope of this line will be the spring constant.

First, we need to tell Sage about the data, using the following command which is an example of how to write lists in Sage and Python:

```
datapoints = [(5, 1.02), (10, 1.86), (15, 3.00), (20, 3.94),
    (25, 4.95), (30, 5.82), (35, 6.95), (40, 7.80)]
```

We note that we ordinarily would not enter points by hand like this because we'd likely have hundreds or thousands of them. We would ordinarily read the points into Sage from a file. We will learn how to do this down the road. For a small list like this, it's fine to enter the points by hand as we did here.

Notice how the data, namely those eight points, are separated by commas and enclosed by brackets. This is an example of a list in Sage. You can enclose any data with `[` and `]`, and separate the entries by commas to make a list.

Now we can easily make a scatter plot by just typing

```
sp = scatter_plot(datapoints)
show(sp)
```

We can certainly use trial and error and our eyeball to find the slope of the best fit line for this data. But there is a mathematical criterion for best fit called *least squares*. This principle gets a line of best fit by using calculus to minimize the sums of the squares of the distances between the real y-coordinates of our data, and the y-coordinates of the points on our proposed line. SageMath can do this to fit a linear model via its built-in `find_fit` command.

```
var('k')
model(x) = k*x
find_fit(datapoints, model)
```

```
[k == 0.19629411764530547]
```

This says that the line of best fit is $y = 0.19629411764530547x$. We can plot this line with our scatter plot.

```
sp + plot(0.19629411764530547*x, (x, 5, 40))
```

We can use this relationship to approximate y-values for x-values within the range of our data. For example, when $x = 13$, we have that y is approximately

```
0.19629411764530547*13
```

```
2.551823529388971
```

Thus, when we have a weight of 13 newtons attached to the spring, the spring will be stretched approximately 2.55 cm.

6.1.1 Inverse Proportionality

Not all pairs of variables are proportional. But some that don't initially appear to have that sort of relationship can be recast to become proportional.

Another well-known proportionality relationship is *Boyle's Law*, which relates the volume of a gas to its pressure at a constant temperature,

$$V = \frac{k}{P}$$

where $V$ denotes the volume of the gas, $P$ denotes the pressure, and k is a constant. To test Boyle's law, a student measures the pressure of a gas at different volumes while keeping the temperature of the gas constant. The resulting data is below.

Table 6.1.2

Volume ($V$)	50	45	40	35	30	25	20	15	10
Pressure ($P$)	27.24	30.36	34.01	38.73	45.5	54.35	68.03	90.53	135.73

We will use this data to verify Boyle's law and find the constant of proportionality for this gas. Note that the law does not say that $V$ is proportional to $P$. It says that $V$ is proportional to $1/P$. Therefore, we will plot $V$ versus $1/P$ and fit a straight line through the origin to this *transformed* data.

```
pvpoints = [(27.24, 50), (30.36, 45), (34.01, 40), (38.73,
    35), (45.5, 30), (54.35, 25), (68.03, 20), (90.53, 15),
    (135.73, 10)]
sp = scatter_plot(pvpoints, axes_labels = ['$P$', '$V$'])
show(sp)
```

This is clearly not linear. We can transform the points as follows:

```
ourpoints = [(1/P, V) for (P, V) in pvpoints]
sp = scatter_plot(ourpoints, axes_labels = ['$1/P$', '$V$'])
show(sp)
```

We can now use a linear model as we did earlier to find the best fit constant k.

```
var('k')
model(x) = k*x
find_fit(ourpoints, model)
```

```
[k == 1361.6466604482955]
```

```
sp + plot(1361.6466604552695*x, (x, 0, 0.04))
```

This shows that $V$ is clearly proportional to $1/P$ and the constant of proportionality is about 1362. (Note that the constant of proportionality may be different for a different gas.) In this case, we say that V and P are *inversely proportional*.

Another way to do this particular model would be to change the model equation and use the original points.

```
var('k')
model(x) = k/x
soln = find_fit(pvpoints, model, solution_dict=True)
```

We've put an extra option on the `find_fit` command this time called `solution_dict`. This records the variables in the model in an internal table so that we don't have to copy and paste long numbers like we did earlier. So if we want to know the value of k that SageMath computed we ask for

```
soln[k]
```

```
1361.6466604652283
```

6.1.2 Exercises

1. For each of the data sets below, determine if it is reasonable to assume that y is proportional to x. If it is, approximate the constant of proportionality. If it is not, describe why this assumption is not reasonable.

(a)

x	1	1.1	1.2	1.3	1.4	1.5	1.6	1.7
y	1	1.22	1.44	1.69	1.96	2.25	2.56	2.89

(b)

x	1	5	7	2	10	12	3	6
y	0.79	10.89	14.37	5.75	23.36	26.29	3.76	16.12

(c)

x	2	6	9	15	7	25	39	4
y	26	20	18	26	6	19	20	13

2. Determine which model best fits the data below. Find the constant of proportionality for each model.

$$y = kx, \quad y = k \cdot \frac{1}{x}, \quad y = kx^2, \quad y = k\sqrt{x}, \quad y = k \cdot \frac{1}{x^3}$$

x	0.5	0.7	0.9	1.2	1.5
y	7.8	3.5	2.2	0.85	0.36

6.2 Modeling with Proportionality

One important observation about a proportionality relationship is that if one of the variables increases, so does the other, and if one variable decreases, so does the other. (We saw this previously in the examples on spring force.) Whenever we encounter a situation where two variables increase or decrease at the same time, we should consider a proportionality relationship.

Example 6.2.1 Population Growth. In many situations involving populations, the larger the population, the faster it grows. This suggests a proportionality relationship between the population and the rate of growth. The table below shows the population (in thousands) of bacteria in a Petri dish at different points in time. The third column contains the change in population between time periods.

Table 6.2.2

Day n	**Actual Population** p_n	**Change in Population** $\Delta p_n = p_{n+1} - p_n$
0	10.3	6.9
1	17.2	9.8
2	27	18.3
3	45.3	34.9
4	80.2	45.1
5	125.3	6.9
6	176.2	79.4
7	255.6	

Observe that as n increases, p_n increases, and so does Δp_n. This suggests that p_n is proportional to Δp_n. A graph of Δp_n vs. p_n is produced with the following commands.

```
poppoints = [(10.3, 6.9), (17.2, 9.8), (27, 18.3), (45.3,
    34.9), (80.2, 45.1), (125.3, 50.9), (176.2, 79.4)]
sp = scatter_plot(poppoints, xmin = 0, xmax = 200, ymin = 0,
    ymax = 100, frame = True, axes_labels = ["Population",
    "Change␣in␣Population"])
sp
```

(Note we can get nice looking axes labels by including the `frame = True` option. Take it out and notice the difference.)

```
var('k')
model(x) = k*x
soln = find_fit(poppoints, model,solution_dict=True)
sp + plot(soln[k]*x, 0, 200)
```

```
soln[k]
```

```
0.4666257644974303
```

We see that the points do fall near a straight line through the origin, suggesting that proportionality is a reasonable assumption. The slope of this line is approximately 0.5. This gives a model that relates the population on one day, p_n, to the population on the next, p_{n+1}:

$$\Delta p_n = p_{n+1} - p_n = 0.5 p_n \quad \Longrightarrow \quad p_{n+1} = 1.5 p_n$$

This model predicts that the population grows by about 50% each time period, which means the population will grow without bound. This seems unreasonable, so the model needs to be refined. We will do just this later in the course. □

Example 6.2.3 Radioactive Decay. One-half of the amount of a radioactive substance decays after each half-life. Radioactive Carbon-14 (^{14}C) has a half-life of 5715 years. If we start with 10 g of ^{14}C, the table below shows the amount of material remaining after each half-life along with the rate of change between time periods.

Time (years)	**Amount**	**Change**
0	10	5
5,715	5	2.5
11,430	2.5	1.25
17,415	1.25	0.625
22,860	0.625	

Note that as the amount of ^{14}C decreases, the rate at which it decreases also changes. This suggests a proportionality relationship between the amount of ^{14}C and the rate at which it decreases.

If we let $y(t)$ represent the amount of ^{14}C at time t, this proportionality relationship gives the differential equation

$$\frac{dy}{dt} = k\,y.$$

That is, the derivative of y is a multiple of y. The exponential function $y(t) = Ce^{kt}$ solves the differential equation, where C is the initial amount of material. □

Example 6.2.4 Free-falling Object. An object in free–fall encounters two basic forces. The first is its weight due to gravity. The second is air resistance which slows the rate of fall. Air resistance is typically negligible at low speeds so it is often not modeled. If we ignore air resistance, then the only force acting on the object comes from acceleration due to gravity. This leads to the simple differential equation

$$\frac{dv}{dt} = g$$

where $v(t) =$ the velocity of the object at time t and $g = 9.8\,\mathrm{m/sec}^2$ (the acceleration due to gravity). Solving this differential equation yields the model $v(t) = gt + v_0$ where v_0 is the initial velocity. This model predicts that the velocity grows without bound, which is inaccurate. Physics tells us that a free-falling object reaches a "terminal velocity" where the deceleration due to air resistance equals the acceleration due to gravity. At this point the object remains at a fairly constant velocity.

To refine this model for velocity, we can incorporate a simple model for air resistance. It seems reasonable to assume that as velocity increases, the force due to air resistance increases. This implies a proportionality relationship between velocity and the force due to air resistance:

$$\text{Force due to air resistance} = kv$$

This force acts upward on the object. There is also a force acting downward on the object due to its mass m:

$$\text{Downward force} = mg$$

Now, by Newton's second law,

$$F = ma = m\,\frac{dv}{dt}$$

Also,

$$F = \text{Downward force} - \text{Upward force}$$

Putting all this together we get,

$$m\frac{dv}{dt} = mg - kv \Longrightarrow \frac{dv}{dt} + \frac{k}{m}v = g$$

Solving this last differential equation gives the model

$$v(t) = \frac{mg}{k}\left(1 - e^{-kt/m}\right)$$

Note that in this model,

$$\lim_{t\to\infty} v(t) = \frac{mg}{k}$$

This suggests a terminal velocity, so this model is more realistic. □

Proportionality relationships satisfy the following transitive property:

Theorem 6.2.5 *If $a \propto b$ and $b \propto c$, then $a \propto c$.*

Proof. By definition, $a \propto b$ and $b \propto c$ mean that $a = k_1 b$ and $b = k_2 c$ for some nonzero constants k_1 and k_2. So, substituting we get

$$a = k_1 k_2 c$$

but $k_1 k_2$ is a nonzero constant, so $a \propto c$ by definition. ■

The next example illustrates an application of this property.

Example 6.2.6 Work Done by a Train. If a constant force is applied to an object moving it some distance, the work done is defined to be

$$\text{Work} = \text{Force} \times \text{Distance}$$

Suppose a train engine pulls a car along a flat stretch of track until it runs out of fuel, and assume that the force needed is constant. We want to model the total work done by the engine in terms of the amount of fuel it carries.

Since the force is constant, work is proportional to the distance pulled. Let W denote work and D denote distance. In standard notation,

$$W \propto D$$

Now, the total distance the train can pull the car is related to the amount of fuel. The more fuel, the farther it can pull the car. So distance pulled is proportional to the amount of fuel. If A denotes the amount of fuel, we have

$$D \propto A$$

Combining these two proportionality relationships, we arrive at the model:

$$W \propto A$$

Although we do not know the constant of proportionality, we can use this relationship to find relative values. For instance, if the constant of proportionality were 10 and the tank holds 1000 gallons, with a full tank the train could perform

$$W = 10(1000) = 10{,}000$$

units of work. If the fuel tank were one–quarter full, it could perform

$$W = 10(250) = 2500$$

units of work, which is exactly one–quarter as much work as if the tank were full. □

Exercises

1. A very simple assumption about the population of rabbits in a forest is that it grows at a rate proportional to the size of the population and the rabbits die at a rate proportional to the number of foxes in the forest. If p_n denotes the population of rabbits at time n, and F denotes the (constant) number of foxes, this assumption yields a model for the change in the rabbit population:

$$\Delta p_n = p_{n+1} - p_n = k_1 p_n - k_2 F$$

where $k_1, k_2 \geq 0$.

(a) Solve this model for p_{n+1}.

(b) If $p_0 = 500$, $k_1 = 0.15$, and $k_2 = 0.25$, algebraically find the values of F for which the population of rabbits is decreasing, for which it is increasing, and for which it is constant for all $n \geq 0$. (*Hint:* If the population is constant, $p_1 = p_0$.)

(c) Use SageMath to numerically and graphically support your answer above.

2. Refer back to the Population Growth example at the start of this section. We will analyze a refined model. Assume that Δp_n is proportional to the product of the population and its difference from 621, that is

$$\Delta p_n = p_{n+1} - p_n = k p_n (621 - p_n)$$

(a) Use the data in the Population Growth example table to test this assumption by plotting Δp_n vs. $p_n(621 - p_n)$. Use the graph to estimate the constant of proportionality.

(b) Starting with $p_0 = 10.3$, use this refined model to predict the population for days 1 through 20.

(c) Does this model seem more or less reasonable than the original one? Why or why not?

3. Refer again back to the Population Growth example at the start of this section, and consider the assumption that Δp_n is proportional to p_n^2. Use the data in the table to test this assumption. Does this assumption appear to be reasonable? Why or why not?

6.3 Fitting Straight Lines Analytically

As we have seen, modeling with proportionality often requires us to fit a straight line to a set of data. In earlier sections we used a graphical approach, which can be rather subjective. Today we will look at different definitions of a "best fit" line and find formulas for the slope and y-intercept of a best-fit line in terms of the x- and y-coordinates of the data points. This will give an objective approach to fitting a straight line.

The first step is to define criteria for a good-fitting line. The line in the left graph in Figure 6.3.1 fits the data "better" than the line in the right graph.

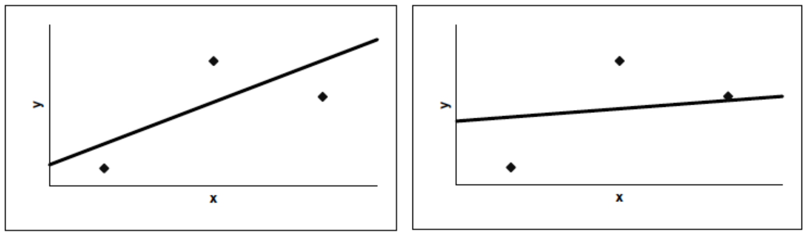


Figure 6.3.1

What's the difference between these two lines? There are probably many ways to answer this question.

We see that in the right graph, the line is very close to the right-most point, but further from the other two points than the line in the left graph. We might say that the line in the left graph is "closer" to the data points in general than the line in the right graph. The idea of minimizing the distance between the line and the points will form the basis of the definition of a best-fit line.

These distances (also called *errors*) are illustrated in the figure below with the dashed lines.

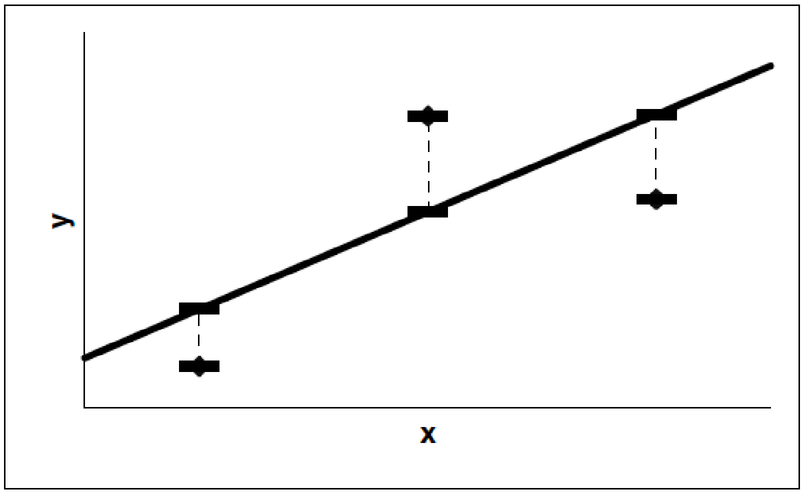


Figure 6.3.2

If the coordinates of the points are given by

$$(x_i, y_i) \text{ for } i = 1, 2, \ldots, n$$

and the line is described by the function $f(x) = mx + b$, then the values of the distances are

$$|y_i - f(x_i)| \text{ for } i = 1, 2, \ldots, n$$

There are many ways to define how the best-fit line minimizes these distances. One way, called *Chebyshev's criterion*, is based on the idea that the best-fit line should make the largest of these distances as small as possible. In more technical terms, this criterion says that the function $f(x) = mx + b$ giving the best-fit line is the one that minimizes the number

$$C = \text{Maximum of } \{|y_i - f(x_i)| : i = 1, 2, \cdots n\}.$$

Our goal is to find formulas for the slope m and y-intercept b. Since we want to minimize something, we might think about using derivatives. Chebyshev's criterion makes logical sense, but it's not obvious how to take the derivative or use it to find simple formulas for m and b.

Another criterion is based on the idea that the best–fit line should minimize the sum of the distances. In mathematical notation, $f(x) = mx + b$ should minimize the number

$$A = \sum_{i=1}^{n} |y_i - f(x_i)|.$$

This criterion is also very logical, but the absolute values make the derivative difficult to calculate. To make the derivative simpler, we might consider getting rid of the absolute values in the above summation altogether and add the criterion that the sum must be non–negative. This, however, would make some of the terms in the summation positive and some negative. So, there might be some large positive values that cancel out some large negative values resulting in a small sum, but a poor-fitting line.

The most widely used criterion, called the *least–squares* criterion, uses squares rather than absolute values to make all the terms positive. In mathematical notation, this criterion says that the function $f(x) = mx + b$ should minimize the number

$$S = \sum_{n=1}^{\infty} (y_i - f(x_i))^2.$$

To see how this might work, consider the example below.

Example 6.3.3 Illustrating the Least-Squares Criterion. We start with a data set and then ask SageMath to produce a scatterplot of the data.

```
datapoints = [(0.8, 2), (2.5, 4.2), (3.5, 3.5), (4.2, 5.3),
    (5.8, 4.5), (7.5, 7)]
sp = scatter_plot(datapoints)
show(sp)
```

We could certainly try to "guess and check" a slope and intercept, but it's likely there's a better candidate than our eyeball might give us.

The least-squares approach to finding a "best-fit" line for a set of data involves finding the slope m and the y-intercept b that minimizes the sum of the squares of the errors. If we let $f(x) = mx + b$ be our approximating line, we are trying to find m and b values so that

$$S = \sum_{i=1}^{n} (y_i - f(x_i))^2 = \sum_{i=1}^{n} (y_i - mx_i - b)^2$$

is a minimum. To search for these values, we think of S as a function of m and b, and set the partial derivatives of S with respect to m and b to 0.

$$\begin{aligned}
\frac{\partial S}{\partial m} &= \sum_{i=1}^{n} 2(y_i - mx_i - b) \cdot (-x_i) = -2\sum_{i=1}^{n} (y_i - mx_i - b)x_i = 0 \\
\frac{\partial S}{\partial b} &= \sum_{i=1}^{n} 2(y_i - mx_i - b) \cdot (-1) = -2\sum_{i=1}^{n} (y_i - mx_i - b) = 0
\end{aligned}$$

These equations can be rewritten as

$$\sum_{i=1}^{n} x_i y_i - m\sum_{i=1}^{n} x_i^2 - b\sum_{i=1}^{n} x_i = 0$$

$$\sum_{i=1}^{n} y_i - m\sum_{i=1}^{n} x_i - n \cdot b = 0$$

Despite the look of these equations, they are a system of two linear equations in two unknowns. Using the usual methods to solve for m and b, we get

$$b = \frac{\sum x_i^2 \sum y_i - \sum x_i y_i \sum x_i}{n \sum x_i^2 - \left(\sum x_i\right)^2}$$
$$m = \frac{n \sum x_i y_i - \sum x_i \sum y_i}{n \sum x_i^2 - \left(\sum x_i\right)^2}$$

These formulas are implemented below.

```
n = len(datapoints)
n
```

```
6
```

```
b = (sum(x[0]^2 for x in datapoints)*sum(x[1] for x in
        datapoints)-sum(x[0]*x[1] for x in datapoints)*
        sum(x[0] for x in datapoints))/(n*sum(x[0]^2 for x in
        datapoints) - sum(x[0] for x in datapoints)^2)

m = (n*sum(x[0]*x[1] for x in datapoints)-
     sum(x[0] for x in datapoints)*sum(x[1] for x in
     datapoints))/(n*sum(x[0]^2 for x in datapoints) -
     sum(x[0] for x in datapoints)^2)

m, b
```

```
(0.632985312334100, 1.85307615171356)
```

The good news is that we don't have to do this all the time. The least-squares approach is already built into SageMath, using the `find_fit` function.

```
var('m1␣b1')
model(x) = m1*x + b1
```

```
sol = find_fit(datapoints, model, solution_dict = True)
sol[m1], sol[b1]
```

```
(0.6329853123708113, 1.8530761494230292)
```

```
sp + plot(sol[m1]*x + sol[b1], (x, 0, 8))
```

□

Exercises

1. Suppose a biologist records the number of pulses per second of the chirps of a cricket at different temperatures (in °F). The data collected is shown below.

Temperature	72	73	89	75	93	85	79	97	86	91
Pulses/sec	16	16.2	21.2	16.5	20	18	16.75	19.25	18.25	18.5

(a) Fit a straight line to this data (where temperature is on the x-axis). How well does the model fit the data?

(b) What is the slope of this line? What does the sign of the slope tell you about the relationship between pulses/sec and temperature?

2. The table below gives the win/loss percentage and other key statistics of 15 NCAA Division 1 women's basketball teams (data collected by Quinn Wragge, 2019).

Win/Loss Percent	**Rebounds Per Game**	**Turnovers Per Game**	**Steals Per Game**	**3 pt Shots Made/Game**	**Field Goal Percent**
93.8	48.8	13.1	7.4	3.3	51.5
52.6	32.89	13.4	4.6	10.3	42.9
89.5	39.89	14.2	10.2	4.8	45.3
77.8	42.67	14.6	8.1	8.9	45.2
68.8	36.19	13.8	8.8	7.7	41.2
100	45.53	14.3	5.6	8	46.3
50	40.82	15.7	7.8	7.2	42.9
94.7	44.17	14.5	8.7	4.1	51.7
68.4	38.26	13.7	7.4	7.5	46.6
70.6	46	16.8	9.7	6.2	42.9
83.3	43.28	16.4	7.6	5.6	45.5
70.6	38.41	15.9	9.6	6.2	43.3
94.1	42.13	11.8	7.3	7.9	49.1
33.3	38.29	16.7	7.4	5.4	39
85	38.84	13.4	9.5	8.2	44.4

(a) Create graphs of win/loss percentage vs. each of the other statistics (one graph per statistic) and fit a straight line to the data.

(b) Comment on how well each line fits the data. Which line appears to fit the data the best?

(c) Comment on the sign of the slope of each line. Does the sign make sense? Briefly explain.

6.4 Geometric Similarity

Shapes such as circles and rectangles are easy to work with. We can calculate the area and volume of objects with these shapes using very simple formulas. Real world objects rarely come in these simple forms. This necessitates some simplifying assumptions. Geometric similarity is one such assumption.

Definition 6.4.1 Two objects are *geometrically similar* if the following two conditions are met:

1. There is a one-to-one correspondence between points of the objects (i.e. the two objects have the same "shape">).

2. The ratio of distances between corresponding points is the same for all pairs of points.

◊

In simpler terms, two objects are geometrically similar if one is a scaled up or down version of the other.

What does geometric similarity allow us to do? Let's start with a very simple example. Consider a rectangle of length 3 cm and height 2 cm. Its area is 6 cm^2. Note that this area is proportional to the square of the length of the rectangle since

$$6 = \frac{6}{3^2}(3^2) = \frac{2}{3}(3^2)$$

Now consider another rectangle of length 5 cm and height 4 cm. Its area is 20 cm^2 which is again proportional to the square of the length since

$$20 = \frac{20}{5^2}(5^2) = \frac{4}{5}(5^2)$$

Consider a third rectangle of length 6 cm and height 4 cm. This rectangle is a scaled up version of the first rectangle (its dimensions are simply 2 times the dimensions of the first one). In other words, these two rectangles are geometrically similar. Its area is 24 cm^2 which is again proportional to the square of the length since

$$24 = \frac{24}{6^2}(6^2) = \frac{2}{3}(6^2)$$

Notice that the constants of proportionality are the same for these two geometrically similar rectangles. The second rectangle is not geometrically similar to the other two rectangles, and its constant of proportionality is not the same as the others.

Let's generalize this example. Suppose we have a rectangle of length $3k$ cm and width $2k$ cm where k is some positive number. This rectangle is geometrically similar to the first rectangle. Its area is $6k^2$ cm^2, which is proportional to the square of the length since

$$6k^2 = \frac{6k^2}{(3k)^2}(3k)^2 = \frac{2}{3}(3k)^2$$

Again note that the constant of proportionality is the same as the other geometrically similar rectangles. What does this mean? Suppose we have a bag of rectangles each of length $3k$ cm and width $2k$ cm where $k > 0$ is different for each rectangle. If we reached into the bag and pulled out one rectangle and wanted to know its area, we wouldn't need to measure both the length and the height. We could simply measure the length, square it, and take it times 2/3. This simplifies the process of finding the area.

The length is an example of a *characteristic dimension*. A characteristic dimension is simply a dimension of the object that is easy to measure. We could have chosen height as the characteristic dimension and done the same analysis as above, but the constant of proportionality would have been different.

This generalization illustrates the first important property of geometrically similar objects.

Theorem 6.4.2 *Suppose H is a set of geometrically similar objects. Let A denote the surface area of an object and ℓ denote a characteristic dimension. Then*

$$A \propto \ell^2,$$

and the constant of proportionality is the same for every object in H.

Example 6.4.3 Wool from a Sheep. A shepherd wants to predict the volume of wool he will get from a sheep, V, in terms of the girth of the sheep (the distance around the fattest part of the belly). The volume is the thickness of the wool times the area from which it is shaved. This area is not a simple shape, so we will use geometric similarity to simplify the model. Consider the following assumptions:

1. The thickness of the wool is the same for every sheep.
2. The area on each sheep from which the wool is shaved is geometrically similar.

The first assumption allows us to model

$$V \propto A$$

where A is the area from which the wool is shaved. The second assumption allows us to model

$$A \propto \ell^2$$

where ℓ is some characteristic dimension. We will choose the girth (the distance around the belly of the sheep) to be this dimension. Combining the two proportionalities using transitivity, we get

$$V \propto \ell^2$$

To find the constant of proportionality, and test the assumptions, we would need to collect data of volume and girth. □

Now let's consider a three-dimensional object. Specifically consider a rectangular box with width 4 cm, height 3 cm, and depth 2 cm. Its volume is 24 cm^3, which is proportional to the height cubed since

$$24 = \frac{24}{3^3}(3^3) = \frac{8}{9}(3^3)$$

Consider a geometrically similar box with width $4k$ cm, height $3k$ cm, and depth $2k$ cm where $k > 0$. Its volume is $24k^3$ cm^3, which again is proportional to the height cubed since

$$24k^3 = \frac{24k^3}{(3k)^3}(3k)^3 = \frac{8}{9}(3k)^3.$$

Note that the constant of proportionality is the same. This generalization illustrates the second important property of geometrically similar objects.

Theorem 6.4.4 *Suppose H is a set of geometrically similar objects. Let V denote the volume of an object and ℓ denote a characteristic dimension. Then*

$$V \propto \ell^3,$$

and the constant of proportionality is the same for every object in H.

As in the first property, no special shape of the objects is assumed. This property allows us to relate the volume of an object to some characteristic dimension, and combining this with the first property we can relate volume to surface area.

Example 6.4.5 Surface Area of a Potato. Suppose we want to fix a large batch of the recipe "Crispy Potato Skins" for an appetizer at our Super Bowl party. This recipe requires only the skin from a potato, so when we buy the potatoes, we want to get the maximum surface area for our money.

At the supermarket we have the choice of several different sizes of potatoes. We have to decide if we want to buy several small potatoes or a few large ones (we are assuming that we can choose individual potatoes). Let's restrict ourselves to the following problem:

Should we buy 8 small baking potatoes weighing 0.25 lbs each, or 4 large baking potatoes weighing 0.5 lbs each?

To answer this question, we want to relate the surface area of a potato, A, to its weight, W. Consider the following assumptions:

1. Potatoes have a constant density.
2. Potatoes are geometrically similar.

The first assumption seems very reasonable. The veracity of the second is arguable. However, potatoes have a very irregular shape, so we need some sort of simplifying assumption to model their surface. Similarity is a *reasonable* assumption.

Now, Weight = density × volume, so the first assumption allows us to model

$$W \propto V$$

where W is the weight and V is the volume. The second assumption allows us to model

$$V \propto \ell^3$$

where ℓ is any characteristic dimension (such as length). Combining the two we get

$$W \propto \ell^3. \tag{6.4.1}$$

The second assumption also allows us to model

$$A \propto \ell^2,$$

where A is the surface area of a potato and ℓ is the same characteristic dimension used in (6.4.1). Rewriting and combining the last two proportionalities, we get

$$\ell \propto W^{1/3} \longrightarrow A \propto \left(W^{1/3}\right)^2 = W^{2/3}$$

Since potatoes are sold by the pound, each choice in the original problem will cost the same amount, so we want the choice with the largest surface area. If A_S and A_L represent the total surface area of the small and large potatoes, respectively, then the last equation gives

$$A_S = 8k(0.25)^{2/3} \qquad \text{and} \qquad A_L = 4k(0.5)^{2/3}$$

where k is some constant (note k is the same for both A_S and A_L). Thus

$$\frac{A_S}{A_L} = \frac{8k(0.25)^{2/3}}{4k(0.5)^{2/3}} \approx 1.26 \quad \Longrightarrow \quad A_S \approx 1.26A_L$$

Therefore, the surface area of the small potatoes is approximately 26% greater than the surface area of the larger potatoes. So we should buy the smaller potatoes. □

Exercises

1. In our earlier example, we assumed potatoes are geometrically similar and of constant density. This yielded the model $W \propto \ell^3$ where W is weight and ℓ is some characteristic dimension. To test these assumptions, a student measured the length (inches) and weight (pounds) of several yellow and Idaho russet potatoes as shown in the table below (data collected by Brennan DeForest, 2019).

Yellow		**Idaho Russet**	
Length	**Weight**	**Length**	**Weight**
2.5	0.2	4.75	0.55
3.5	0.4	5.5	0.55
3	0.3	5	0.7
2.75	0.2	5.25	0.7
3.5	0.4	4.75	0.4
3.25	0.35	5.5	0.7
2.5	0.25	5.75	0.75
3	0.3	5	0.55
3.25	0.4		
3.25	0.4		
3.25	0.35		
2	0.25		

(a) Use the data to determine if the model $W \propto \ell^3$ is reasonable for the yellow potatoes.

(b) Repeat the previous part for the Idaho russet potatoes.

(c) Combine the two types of potatoes into one large sample and repeat part (a).

(d) What do these results suggest about the validity of the assumptions?

2. The table below gives the overall length (inches) and weight (pounds) of several male black bears (data collected by Brent Troyer, 2011). If we assume male black bears are geometrically similar, then we would expect that Weight $\propto$ Length^3. Use this data in the table to determine if geometric similarity is a reasonable assumption.

Length	138	166	180	129.5	150	132	148	140	137	149
Weight	60	155	220	105	110	75	105	90	75	115
Length	102	173	104.5	138	144.5	164	129	158	150	142
Weight	35	220	33	90	80	180	77	120	100	100

6.5 Linearizable Models

In previous sections we used theory of one form or another to construct models and then used data to determine the values of parameters within the model. This process is called **model fitting** and the resulting models are called **analytical models**. The model never fit the data perfectly, but we were willing to accept some error because the model helps *explain* the behavior of the system.

In this section, and the next chapter, we build models guided solely by data. We will not even attempt to use theory to explain behavior. Rather, we will find a model that captures the trend of the data and use it to *predict* values rather than explain the behavior. These models are called **empirical models**. Many of the topics related to empirical modeling are closely related to the field of statistics, and particularly the topic of regression.

Linearizable models are those which can be fit to a set of data by making an appropriate transformation and then fitting a linear model to the transformed data. Listed below are the three common types of linearizable models:

Logarithmic	**Power**	**Exponential**
$y = a + b\ln(x)$	$y = ax^b$	$y = ae^{bx}$

The variable x is called the **predictor** variable while the variable y is called the **response** variable. Graphs of these different types of models are shown in Figure 6.5.1.

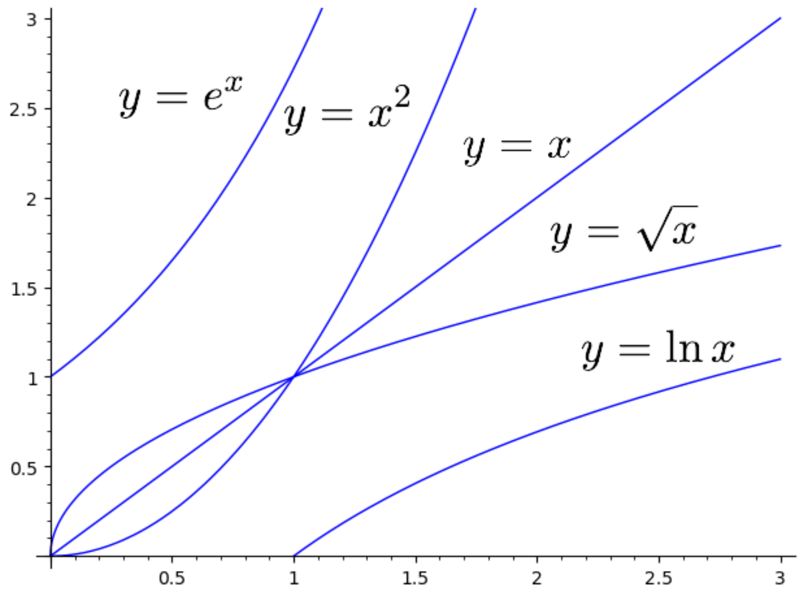


Figure 6.5.1

Note the "shape" of the different graphs. Each one of these different functions increases as x increases, but they increase at different rates. The logarithmic function and the power functions with exponents less than 1 increase much slower than the other types of functions. They almost appear to "level off" whereas the other types grow very quickly. Being able to recognize the shapes of the different graphs will help us to select an appropriate type of model.

To illustrate how to use these models, consider the data in the table below which gives the number of people per physician and male life expectancy (in years) for various countries around the world (data from *World Almanac Book of Facts*, 1992, Pharos Books). Our goal is to predict life expectancy in terms of the number of people per physician.

Country	**People/Physician** P	**Life Expectancy** $L$
Spain	275	74
United States	410	72
Canada	467	73
Romania	559	67
China	643	68
Taiwan	1010	70
Mexico	1037	67
South Korea	1216	66
India	2471	57
Morocco	4873	62
Bangladesh	6166	54
Kenya	7174	59

Obviously there are many factors involved with life expectancy; the number of persons per physician is only one of them. It seems reasonable to believe that as the number of persons per physician increases (meaning fewer doctors per person), life expectancy decreases because people would not have as easy access to health care. We do not claim that the number of people per physician *causes* life expectancy, but there is a relationship between the two variables. It is not at all clear how the variables of people per physician and life expectancy are related theoretically, so we will not even attempt to construct an analytical model. We will construct an empirical model by fitting various linearizable models to this data and analyzing how well each one fits.

We first enter the data points:

```
datapoints = [(275, 74), (410, 72), (467, 73), (559, 67),
        (643, 68), (1010, 70), (1037, 67), (1216, 66),
        (2471, 57),(4873, 62), (6166, 54), (7174, 59)]
```

```
sp = scatter_plot(datapoints)
show(sp, axes_labels = ['People/Physician', 'Life␣
    Expectancy'],figsize = [8,4], frame = True)
```

Notice that as the number of people per physician increases, the life expectancy decreases, agreeing with our intuition. Also note that the points seems to form a curve that initially decreases rapidly, but then levels off. This suggests that a logarithmic or power model might be appropriate.

6.5.1 Logarithmic model

We will fit a curve of the form $L = a + b\ln(P)$ by graphing $L$ versus $\ln(P)$ and fitting a straight line. The value of b is the slope of the line and the value of a is the y-intercept.

```
logpoints = [(ln(P), L) for (P,L) in datapoints]
sp2 = scatter_plot(logpoints, title = "Transformed␣Data",
        axes_labels = ["$\\ln(P)$", "$L$"], frame = True)
show(sp2)
```

This data looks to be more along a straight line than our original data set, so we can then ask Sage to fit a straight line to it.

```
var('b␣a')
model(x) = b*x + a
sol = find_fit(logpoints,model, solution_dict = True)
sol
```

```
{a: 103.40031193527923, b: -5.287622670252565}
```

So the model is $L = 103.4 - 5.2876 \ln P$.

```
sp2 + plot(sol[b]*x + sol[a], (x, 5, 9) )
```

Now we want to create a graph to compare the observed values of L to the predicted ones from our model.

```
predpoints = [(P,sol[b]*ln(P) + sol[a]) for (P,L) in
    datapoints]
show(sp + list_plot(predpoints, size=75, marker='v'),
     axes_labels = ["People/Physician", "Life␣Expectancy"],
     frame = True, xmin = 0, xmax = 8000, ymin = 50, ymax =
         80)
```

To further analyze how well the model fits the data, for each data point define

$$\text{Residual} = (\text{Observed value}) - (\text{Predicted value}).$$

Note that a positive residual means that the predicted value is less than the observed value. A negative residual means that the predicted value is greater than the observed value.

```
respoints = [(x, y-(sol[b]*ln(x) + sol[a]))
        for (x,y) in datapoints]
list_plot(respoints, size=75, title = "Residuals",
        axes_labels = [None, "Residuals"], frame = True)
```

Note that roughly half the residuals are positive and half are negative. This indicates that the model does not tend to overpredict or underpredict the values of L. Also note that the magnitudes of the residuals (the absolute values) are all relatively small, less than 6, and that there is no "pattern" to the residuals. These three observations indicate that this model fits relatively well.

We note that we can find the logarithmic model directly by using that logarithmic format in our definition of `model(x)` and applying it directly to the data points themselves (not the logarithm version).

```
var('b␣a')
model(x) = b*ln(x) + a
find_fit(datapoints, model)
```

6.5.2 Power model

We will fit a curve of the form $L = aP^b$ to the data. To find the values of this, we linearize the model by taking the natural logarithm of both sides of its equation to get

$$\ln L = \ln(aP^b) = \ln a + \ln P^b = \ln a + b \ln P.$$

Thus a straight line fit to the graph of $\ln L$ versus $\ln P$ will have a slope of b and a y-intercept of $\ln a$.

```
loglogpoints = [(ln(P), ln(L)) for (P,L) in datapoints]
sp3 = scatter_plot(loglogpoints, title = "Transformed␣Data",
        axes_labels = ["$\\ln(P)$", "$\\ln(L)$"], frame =
            True)
show(sp3)
```

```
var('b␣a')
model(x) = b*x + a
sol = find_fit(loglogpoints, model, solution_dict = True)
sol
```

```
{a: 4.767541100938781, b: -0.08233961622369507}
```

The equation of the line through the log-log data is $y = -0.0823x + 4.7675$.

```
sp3 + plot(sol[b]*x + sol[a], (x, 5, 9))
```

Therefore, we have

$$\ln a = 4.7675 \quad\Longrightarrow\quad a = e^{4.7675} = 117.62.$$

Therefore our model is $L = 117.62P^{-0.0823}$. Again, we could have computed this directly from the original data points.

```
var('a␣b')
model(x) = a*x^b
sol = find_fit(datapoints, model, solution_dict = True)
sol
```

```
{a: 117.64062968551121, b: -0.08223680619371171}
```

```
predpoints = [(P, sol[a]*P^sol[b] ) for (P,L) in datapoints]
sp + list_plot(predpoints, size=75, marker='v')
```

Now plot the residuals and determine whether we have a good model.

```
respoints = [(P,L-sol[a]*P^sol[b] ) for (P,L) in datapoints]
list_plot(respoints, size=75,
        title = "Power␣Model␣Residuals",
        axes_labels = [None, "Residuals"], frame = True)
```

6.5.3 Exponential model

Now try and fit a model of the form $L = ae^{bP}$ to the data, plot the residuals and tell whether the model is a good fit.

Note that

$$\ln(L) = \ln(ae^{bP}) = \ln a + \ln(e^{bP}) = \ln a + bP.$$

```
exppoints = [(P,ln(L)) for (P,L) in datapoints]
sp4 = scatter_plot(exppoints,
        axes_labels=["$P$", "$\\ln␣L$"],
    frame=True)
sp4
```

```
var('a␣b')
model(x) = b*x+a
sol = find_fit(exppoints, model, solution_dict=True)
sol
```

```
{a: 4.256131024464953, b: -3.416720158289088e-05}
```

The equation of the line through the transformed data is $y = -0.00003x + 4.2561$, so $b = -0.00003$ and

$$\ln a = 4.2561 \Longrightarrow a = e^{4.2561} \approx 70.53$$

Therefore the model is $L = 70.53e^{-0.00003P}$.

```
sp4 + plot(sol[b]*x+sol[a],(x,0,8000))
```

As before, we will graph our predicted values with the actual data points, and compute the residuals to check fit.

```
predpoints = [(P, e^sol[a]*e^(sol[b]*P) ) for (P,L) in
    datapoints]
sp + list_plot(predpoints, size=75, marker='v')
```

```
respoints = [(P,L-e^sol[a]*e^(sol[b]*P) ) for (P,L) in
    datapoints]
list_plot(respoints, size=75,
        title = "Exponential␣Model␣Residuals",
        axes_labels = [None, "Residuals"], frame = True)
```

Notice that all the residuals are at least 1 in magnitude and that one is almost -8. This indicates that the model doesn't predict any of the values of L very accurately. Therefore, this is not the best fitting model, as previously suspected.

We note that in this case, we could *not* have computed the exponential model directly.

```
var('a␣b')
model(x) = a*e^(b*x)
find_fit(datapoints, model)
```

```
[a == 1.0, b == 1.0]
```

The `find_fit` has some issues when working on exponential models. So when computing an exponential model in SageMath, *we will most likely have to linearize the model before using* `find_fit`.

6.5.4 Using the models to predict

Out of the three models, the logarithmic and power models are the "best" based on an analysis of the graph of the models and of the residuals. In the next section we will look at more analytical techniques for measuring how well a model fits a set of data.

Now that we have two good–fitting models for the data, what can we do with them? There are at least two uses. First of all, the graphs of the models help us to see the trend of the data. The graphs decrease from left to right, helping us to illustrate the point that as the number of people per physician increases, life expectancy decreases. The plot of the data also shows this, but a curve helps exemplify the relationship.

Second of all, we can use the models to *predict* life expectancy if we know the number of people per physician. For instance, suppose a country has 3,500 people per physician. The logarithmic model predicts that life expectancy is

```
103.40031225765938 -5.287622715527743*ln(3500.)
```

```
60.2505706018187
```

while the power model gives

```
117.64063035437424*(3500.)^(-0.0822368070118513)
```

```
60.1318413722959
```

We certainly could plug $P = 3,500$ into the exponential model and get

```
70.5365506541459*e^(-3.416720158289088e-05*3500)
```

```
62.5862633626265
```

However, we saw that the exponential model did not fit the data very well, so it would not be appropriate to use it for making predictions. This illustrates the first caution when using empirical models: *If a model does not fit a set of data, do not use it for making predictions.*

Now suppose a country has 100 persons per physician. We could plug $P = 100$ into the logarithmic model to get

```
103.40031225765938 -5.287622715527743*ln(100.)
```

```
79.0499097733576
```

However, note that $P = 100$ is outside the range of the original data. We do not know the trend of the data for values of P less than 275. It could change or stay the same; we simply do not know. Therefore, it would be inappropriate to use any of these models to predict values of L for P less than 275. This value of 79.04 years may or may not be accurate, so we should not report this "prediction." This illustrates the second caution when using empirical models: *Only use values of the predictor variable that are within the range of the original set of data.*

6.5.5 Exercises

1. The table below contains the total length and weight of 20 black bears (data collected by Brent Troyer, 2011). Graph weight vs. length, fit different linearizable models to the data, and select the one that best fits the data. Briefly explain your reasoning.

Length	139	138	139	120.5	149	141	150	166	180	129.5
Weight	110	60	90	60	85	95	85	155	220	105
Length	150	142	162	148	140	134	137	149	102	151.5
Weight	110	115	255	105	90	75	75	115	35	140

2. Consider the data below:

x	0	2	4	6	8	10
y	3.000	3.061	3.122	3.186	3.250	3.316

(a) Fit a linear model to the data. How well does the model appear to fit the data? Create a graph of the residuals. What do you notice about the pattern of the residuals?

(b) Fit an exponential model to the data. How well does the model appear to fit the data? Calculate and graph the residuals. What does this tell you about how well this model fits the data?

6.6 Coefficient of Determination

The coefficient of determination, denoted R^2, is a numerical measure of how well a line fits a set of data. To illustrate the fundamental concepts, we will generate some hypothetical data according to the relationship $y = 3 + 2x$.

```
data = [(x,2*x+3) for x in range(10)]
data
```

```
[(0, 3),
 (1, 5),
 (2, 7),
 (3, 9),
 (4, 11),
 (5, 13),
 (6, 15),
 (7, 17),
 (8, 19),
 (9, 21)]
```

```
var('x') # x had a value of 9 because of our data
    definition.  This resets x and is necessary.
sp = scatter_plot(data)
sp + plot(2*x+3,(x,0,10))
```

Note that the line goes through each data point and the equation of this line (also called the *regression equation* is exactly what is used to generate the data.

One purpose of fitting a line to data is to use it for predicting the value of y when x is known. If we did not have a graph of the data or the regression equation and we were given a value of x, the best guess as to the corresponding value of y would be the mean of the data.

```
yvalues = [y for (x,y) in data]
yvalues
```

```
[3, 5, 7, 9, 11, 13, 15, 17, 19, 21]
```

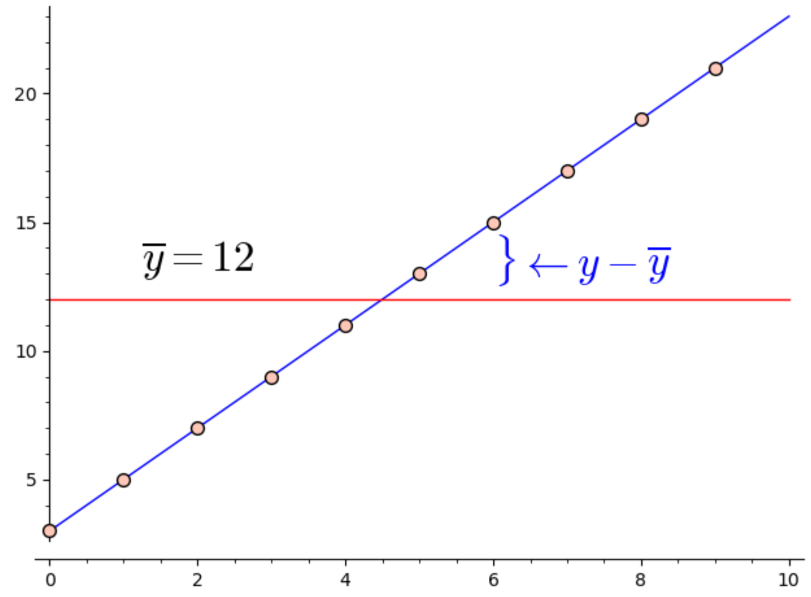
```
import numpy # This imports a numerical package

numpy.mean(yvalues)
```

```
12
```

This mean is denoted by $\overline{y}$ and equals 12 in this case. We test this simple prediction strategy by examining the "error" it would cause for the given data points.

```
var('x')
sp + plot(2*x+3,(x,0,10)) + plot(12,(x,0,10),color='red') \
+ text('$\\}\\leftarrow␣y␣-\\overline{y}$',
    (7.2,13.5),fontsize='24') \
+ text('$\\overline{y}=12$', (2,13.5), fontsize = '24',
    color = 'black')
```



If a y-value predicted by the regression equation is denoted $\hat{y}$, we measure how well a regression equation fits the data by comparing the deviation to the difference between $\hat{y}$ and $\overline{y}$, $\hat{y}-\overline{y}$. In this case, the regression line goes through each data point, so $\hat{y}=y$. Therefore,

$$\hat{y}-\overline{y}=y-\overline{y} \quad \Longrightarrow \quad \frac{\hat{y}-\overline{y}}{y-\overline{y}}=1.$$

Thus we give this regression equation an R^2 value of 1. This value is often interpreted by saying that the regression equation "explains" 100% of the deviation. The definition of the coefficient of determination is based on this idea of comparing the deviation to the difference between $\hat{y}$ and $\overline{y}$ to measure the percentage of deviation "explained" by the regression equation.

This set of data is highly idealized because it was generated exactly according to the linear relationship $y=3+2x$. In reality, data never conforms to an exact relationship like this. Real data with a linear relationship satisfies an equation of the form

$$y=\beta_0+\beta_1 x+\varepsilon$$

where ε is some "noise." This noise may be due to measurement error, sampling variation, or some other random event outside of our control.

The relation $y=\beta_0+\beta_1 x$ is called the "true" linear trend of the population while the regression equation has the generic form $\hat{y}=\hat{\beta}_0+\hat{\beta}_1 x$. The "hats" indicate that the parameters $\hat{\beta}_0$ and $\hat{\beta}_1$, and the values of y, are estimates of the population values based on sample data.

Let's redo the data set by adding in some psuedorandom noise.

```
data = [(x, 2*x+3+numpy.random.normal(0,2)) for x in
    range(10)]

sp2 = scatter_plot(data)
show(sp2)
```

(If you execute these commands several times in a row, you will get a slightly different graph each time.)

```
var('b0␣b1␣x')
model(x) = b0 + b1*x
ans = find_fit(data,model, solution_dict = True)
ans
```

```
sp2 + plot(ans[b0] + ans[b1]*x,(x,0,10))
```

How do we define the R^2 value for a line fit to noisy data? We introduce three different types of deviation:

- *Explained deviation* $= \hat{y} - \overline{y}$
- *Unexplained deviation* $= y - \hat{y}$
- *Total deviation* = Explained deviation + unexplained deviation $= y - \overline{y}$

We wish to total up the deviation over all the data points, but we need to avoid cancellation. So similar to the least-squares approach, we will square these quantities to get amounts of **variation**. The total variation is called the **total sum of squares** and is given by

$$SS_{Tot} = \sum (y_i - \overline{y})^2.$$

The explained variation is called the **regression sum of squares** and is given by

$$SS_{Reg} = \sum (\hat{y}_i - \overline{y})^2.$$

The unexplained variation is called the **residual sum of squares** and is given by

$$SS_{Res} = \sum (y_i - \hat{y}_i)^2.$$

Similar to the relationship between the deviations, we get that

$$SS_{Tot} = SS_{Reg} + SS_{Res}.$$

(Proving that is not trivial.). If we rewrite the previous equation as $SS_{Reg} = SS_{Tot} - SS_{Res}$, we can write that the "percentage" of the total variation explained by the regression equation is

$$R^2 = \frac{SS_{Reg}}{SS_{Tot}} = \frac{SS_{Tot} - SS_{Res}}{SS_{Tot}}.$$

This is the definition of the coefficient of determination. The closer R^2 is to 1, the better the line fits the data. An R^2 value close to 0 indicates a very poor-fitting model.

Now let's calculate R^2 for our noisy data and its regression line above.

```
yvalues = [y for (x,y) in data]
avg = numpy.mean(yvalues)

yhatvalues = [(ans[b0] + ans[b1]*x) for (x,y) in data]

sstot = sum((yvalues[i]-avg)^2 for i in range(10))

ssres = sum((yvalues[i] - yhatvalues[i])^2 for i in
    range(10))

rsquared = (sstot - ssres)/sstot
rsquared
```

The R^2 value is not 1 because our data has some noise, so the underlying linear relationship ($y = 3 + 2x$) does not account for all of the variation from the mean.

If we re-execute the cells above to re-do the noise, we should get different points and a different R^2 value. We can add more "noise" to the data by increasing the standard deviation from 2 to 4. What happens to the R^2 value if we do this?

6.6.1 Finding R^2 for linearizable models

Consider the linearizable models fit to the life expectancy data in Section 6.5. We can compare how well the different models fit the data by calculating the R^2 value for each model and then comparing the R^2 values. To calculate the R^2 value for a linearizable model, we calculate the R^2 value for the straight line fit to the *transformed* data.

6.6.1.1 Power model

```
datapoints = [(275, 74), (410, 72), (467, 73), (559, 67),
    (643, 68), (1010, 70), (1037, 67), (1216, 66), (2471,
    57), (4873, 62), (6166, 54), (7174, 59)]

loglogpoints = [(ln(P), ln(L)) for (P, L) in datapoints]
sp = scatter_plot(loglogpoints, axes_labels=['$\\ln_P$',
    '$\\ln_L$'])
show(sp)
```

```
var('b_a')
model(x) = b*x + a
ans = find_fit(loglogpoints, model, solution_dict=True)
ans
```

```
sp + plot(ans[b]*x + ans[a],(x,5,9))
```

```
yvalues = [y for (x,y) in loglogpoints]

import numpy
avg = numpy.mean(yvalues).n()
avg
```

```
yhatvalues = [(ans[b]*x + ans[a]) for (x,y) in loglogpoints]
sstot = N(sum((yvalues[i]-avg)^2 for i in range(10)))
```

```
ssres = N(sum((yvalues[i] - yhatvalues[i])^2 for i in
    range(10)))
rsquared = (sstot - ssres)/sstot
rsquared
```

```
0.761061535668576
```

6.6.1.2 Logarithmic model

```
datapoints = [(275, 74), (410, 72), (467, 73), (559, 67),
    (643, 68), (1010, 70), (1037, 67), (1216, 66), (2471,
    57), (4873, 62), (6166, 54), (7174, 59)]

logpoints = [(ln(P), L) for (P,L) in datapoints]
sp2 = scatter_plot(logpoints, axes_labels = ["$\\ln␣P$",
    "$L$"])
show(sp2)
```

```
var('b␣a')
model(x) = b*x + a
ans = find_fit(logpoints,model, solution_dict = True)
ans
```

```
sp2 + plot(ans[b]*x + ans[a],(x,5,9))
```

```
yvalues = [y for (x,y) in logpoints]

import numpy
avg = numpy.mean(yvalues)
avg
```

```
yhatvalues = [(ans[b]*x + ans[a]) for (x,y) in logpoints]
sstot = N(sum((yvalues[i]-avg)^2 for i in range(10)))
ssres = N(sum((yvalues[i] - yhatvalues[i])^2 for i in
    range(10)))
rsquared = (sstot - ssres)/sstot
rsquared
```

```
0.771445233869118
```

6.6.1.3 Exponential model

```
datapoints = [(275, 74), (410, 72), (467, 73), (559, 67),
    (643, 68), (1010, 70), (1037, 67), (1216, 66), (2471,
    57), (4873, 62), (6166, 54), (7174, 59)]

exppoints = [(P,ln(L)) for (P,L) in datapoints]
sp3 = scatter_plot(exppoints,axes_labels=["$P$", "$\\ln␣L$"])
sp3
```

```
var('b␣a')
model(x) = b*x + a
ans = find_fit(exppoints,model, solution_dict = True)
ans
```

```
sp3 + plot(ans[b]*x + ans[a],(x,0,7000))
```

```
yvalues = [y for (x,y) in exppoints]

import numpy
avg = numpy.mean(yvalues).n()
avg
```

```
yhatvalues = [(ans[b]*x + ans[a]) for (x,y) in exppoints]
sstot = N(sum((yvalues[i]-avg)^2 for i in range(10)))
ssres = N(sum((yvalues[i] - yhatvalues[i])^2 for i in
    range(10)))
rsquared = (sstot - ssres)/sstot
rsquared
```

```
0.573865088823950
```

We can now compare how well these models fit the data by comparing their R^2 values:

Logarithmic	**Power**	**Exponential**
0.7714	0.7611	0.5739

We see that the power and logarithmic models fit the data well with the logarithmic model being slightly better. The exponential model does not fit as well. Thus, based strictly on the R^2 values, we would conclude that the logarithmic model fits the data the best. This agrees with our earlier conclusion.

We warn against blindly using R^2 values to choose a best model. These values should be used as only one factor when choosing a best model. Other factors that should be considered are the nature of the behavior being analyzed and the simplicity of the model.

For instance, population growth is often exponential. So if we fit curves to some data of a population, we may want to favor an exponential model over other types even if it has a lower R^2 value.

In "Modeling the U.S. Population" (AMATYC Review, Vol. 20, No. 2, Spring 1999, pages 17–29), Sheldon Gordon makes the point that, "The best choice (of a model) depends on the set of data being analyzed and requires an exercise in judgement, not just computation."

6.6.2 Exercises

1. The data below contain the weights (in lbs) and highway miles per gallon (MPG) of several cars. Calculate SS_{Reg}, SS_{Res}, SS_{Tot}, and R^2 for the linear regression equation fit to this data.

Weight (x)	3250	3425	2400	2250
MPG (y)	26	28	37	38

2. The data below contain the diameter of the trunk at chest height and volume of wood in several pine trees. Model Volume in terms of Diameter with several different linearizable models and select the best one. Briefly explain how you decided which one is best.

Diameter	32	29	24	45	20	30	26	40	24	18
Volume	185	109	95	300	30	125	55	246	60	15