Sage for Undergraduates (online version)

Gregory V. Bard

Dept. of Mathematics, Statistics, and Computer Science, University of Wisconsin—Stout, Menomonie, Wi, 54751
E-mail address: bardg@uwstout.edu

2010 *Mathematics Subject Classification.* Primary: 15-04, 34-04, 65-04, 90-04, 97M10. Secondary: 11-04, 12-04, 28-04, 40-04, 68U05.

With warmth I dedicate this book to my husband, Patrick, who has supported me throughout my career, in good years and in bad years, and who has spent many evenings alone because I was working on this book.

Contents

Preface—How to Use This Book

As the open-source and free competitor to expensive software like Maple, Mathematica, Magma and Matlab, Sage offers anyone with access to a web-browser the ability to use cutting-edge mathematical software, and display one's results for others, often with stunning graphics. I'm sure that you will find Sage far easier to use than a graphing calculator, and vastly more powerful.

There is no need to read this entire document, just as you would never read the dictionary cover to cover. This book is ideal for self-study, but if you are assigned this book as part of a class, then your professor will tell you which section numbers correspond with what you need to be reading.

For other readers, who reading independently of a specific class, I have the following suggestions:

- Chapter 1 contains the basics—what you really need to know. Most of the common tasks in Sage were carefully designed to be as intuitive as possible, with notation as close as possible to how mathematics is done on the whiteboard or with a pencil. I recommend that you never try to read more than 3 entire sections in one day. Otherwise there is too much for your brain to absorb while still keeping the experience fun and new. However, 1–2 sections per day should be easily digestible.

Note: Personally, I recommend just reading Chapter 1, and then start playing around on your own. The best way to learn a new bit of

computer software is to use it recreationally. Using the table of contents (near the front of the book) or the index of commands (at the end of the book) you can always look up how to do some task which you haven't learned yet. If you get flummoxed at any point, be sure to check out Appendix A, "What to Do When Frustrated."

- Chapter 2 contains projects where you can use Sage to solve mathematics problems as they arise in other subjects—cryptography, physics, chemistry, biology, and so forth. These projects are primarily aimed to be tractable after reading Chapter 1. However, in some cases, another section from a higher numbered chapter will be required also. Of course, in those cases, the dependency is clearly marked. These projects are intended to be assigned over weekends. They are not large enough to be semester projects, but they are too large to be homework problems.
- Chapter 3 is all about making beautiful plots and graphs. I have put in there everything that I could. However, some types of plots, including 3D plots, animated plots, and multicolored plots, cannot be easily represented in a printed black-and-white book. Therefore, those types of plots are covered in the electronic-only online appendix to this book "Plotting in Color, in 3D, and Animations," available on my webpage `www.gregorybard.com` for downloading.
- Chapter 4 is enormous, and contains small sections that discuss advanced mathematical topics. Sage can handle an enormous variety of math problems. This chapter was designed so that the individual sections can be read independently. There is no need to read the sections in Chapter 4 in order, unless you want to.
- Chapter 5 is about how to program in Python using Sage. As a computer algebra system, Sage is built on top of the computer language Python. Therefore, you can use Python commands in Sage. Sometimes this is really useful—for example, you can write `for` loops to make tables very easily. At other times it is convenient to write entire computer programs in Python using Sage as the interface. This is even more true when using SageMathCloud.
- Chapter 6 covers making interactive webpages (called apps) using Sage. The process is remarkably straightforward. I have a six-stage process which I've been using myself. You can make extremely interesting interacts using Sage, and I'll bet you'll be surprised at how easy it is to make your own.
- Appendix A is "What to Do When Frustrated," where I have a list of questions you should ask yourself if you get stuck in a situation where Sage is refusing to respond to your commands in the way that you'd like. In almost all cases, one of the listed items will present you with a way of resolving the frustration.
- Appendix B is about gaining familiarity with SageMathCloud and some of its features. Cloud computing is a new and exciting way

to compute using the internet, instead of your local machine, as a place for storing your files.

Note: Generally users will want to switch to SageMathCloud after completing Chapter 1, before tackling the projects in Chapter 2 or the advanced work of Chapters 4–6. I think Chapter 3 can be learned before or after learning SageMathCloud, with no disadvantage to the user.

- Appendix C is a collection of further resources on Sage, for readers who would like to learn more than this book contains.
- Appendix D is my response to the fact that students often do not have facility with systems of linear equations that have an infinite number of solutions. This concept is often grasped as an idea, but the details are often not there when I need students to know them. Therefore, I have decided to be hyper-thorough and devote a large number of pages to explaining this (admittedly difficult) subject. I hope it will be useful to students.
- Appendix E is what you should read if you want to learn how to install Sage on your local machine.
- Appendix F is a sixteen (!) page index of all the commands in this book, for your reference. Special thanks to Thomas Suiter, who prepared it.
- The online electronic appendix will cover plotting in color, animations, and 3D graphics. Those subjects are not suited to a black-and-white book, and therefore cannot be printed within these covers. You can download that from my webpage

  `www.gregorybard.com`

  by clicking on the link, at the left of the screen, for Sage.

Let's Dive In! Right Now!

Without any hesitation whatsoever, open up a web browser at this very moment. Type in the following URL:

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

You have now connected to "The Sage Cell Server." Now you shall see a large window (called a cell) and a big button marked "Evaluate." In that cell, type the following

```
solve( x^2 - 16 == 0, x )
```

and then click "Evaluate."

You should receive back the answer

```
[x == -4, x == 4]
```

which is clearly the correct set of two solutions to the equation $x^2 - 16 = 0$. You have now solved your first math problem with Sage. Congratulations.

Welcome to Sage! You are now ready to begin reading Chapter 1.

Acknowledgements

First, I would like to thank William Stein, the creator of Sage, for his encouragement, assistance, and access to supercomputing resources which have made my cryptanalytic research possible. Prof. Stein is the founder of Sage, its chief architect and organizer, and the source of enthusiasm for the Sage community—which by this point now includes hundreds of people across many countries and continents. Many of the original lines of Sage itself, as well as the vast majority of SageMathCloud, were personally coded by Prof. Stein.

This book began as "A Guide to Sage" while I was teaching for five weeks in Beijing at the Chinese Academy of Sciences, Academy of Mathematics and Systems Science, Key Laboratory for Mathematics Mechanization, during the Summer of 2010. I am very grateful for being given that special opportunity to work with such advanced students. I was teaching from another book that I have written, *Algebraic Cryptanalysis*, published by Springer in 2009. I gave several hours of instruction in Sage, but it became clear that having written instructions would be more useful for students. One can thumb through them while working on Sage, and therefore the document that would eventually become this book was born. I am extraordinarily grateful for the opportunity to have visited that excellent institution. When I left there, this document was 37 pages, but from tiny acorns do mighty oak trees grow.

Like many researchers who invent a new algorithm as part of their PhD dissertation at the University of Maryland, I had to write code to prove that my algorithm is effective, correct, and efficient. However, such code is often discarded after the PhD is conferred. I would like to thank Martin Albrecht, who first introduced me to the Sage community and who lead to the inclusion of my dissertation code into the Sage libraries, in what has become the `M4RI` project. Without Martin, I'd never had heard of Sage, nor would my dissertation code be in use by people scattered about the world. As it turns out `M4RI` is an evolving project, continually upgraded by many different researchers, and probably today contains very few of my original

lines. That is an example of how software can evolve. You can read about the original algorithm in Chapter 9 of *Algebraic Cryptanalysis*.

I would also like thank Robert Beezer and Robert Miller for giving me encouragement in using Sage in my teaching and research, respectively. In particular, Robert Beezer's outstanding (and free) eBook *A First Course in Linear Algebra* was an inspiration for me while writing this book.

The Sage Cell Server has made Sage accessible to a much wider audience than has ever before been dreamed possible. Even for those experienced with Sage, the user experience is now vastly improved. This important and difficult work has been done by Jason Grout, Ira Hanson, Steven Johnson, Alex Kramer, and William Stein. Originally, Sage was suitable for graduate students, faculty, and perhaps senior math majors. Then the Sage Notebook Server came, bringing Sage to the mid-career undergraduate. Now, thanks to the Sage Cell Server, it is now appropriate to use Sage even in *Precalculus* classes, if desired.

The American Mathematics Society was most flexible with me on the matters of deadlines as well as the many images in this textbook. Producing a book requires a surprisingly large number of steps, and I am grateful for the support of Ina Mette, the acquisitions editor, and Marcia Almeida, her assistant. I am particularly grateful for Ina Mette's energetic and enthusiastic encouragement that I turn what was once a small guide into a larger (published) book. This project would not exist without her encouragement.

I would like to thank Andrew Novocin for teaching me how to include `*.png` files in LaTeX documents, which saved me the time of manually converting the file formats of the armada of screen captures that fill this book. I might have otherwise abandoned this entire book project.

No author can proofread his or her own work effectively, but that is particularly the case for me. I am very grateful for the assistance of the proofreaders Joseph Robert Bertino, Joseph Loeffler, and Thomas Suiter, who have each made suggestions and corrections on portions of this document. Of particular merit, Thomas Suiter made the index, and Joseph Bertino has nobly borne (without complaint) an enormous workload from me through the last few years—both from this book and other projects. Of course, any errors that remain are my responsibility.

My first job after getting my PhD from the University of Maryland was at Fordham University, in The Bronx, NY, where I was appointed to one of the four-year "Peter M. Curran Visiting Assistant Professorships." I made some progress on this project during my time at Fordham, after returning from my summer in China. I received encouragement and suggestions from Shaun Ault, Melkana Brakalova, Armand Brumer, Michael Burr, Bill Campbell, Janusz Golec, Maryam Hastings, William Hastings, Nick Kintos, Robert Lewis, Damian Lyons, Ian Morrison, Leonard Nissim, Cris Poor, Benjamin Shemmer, Shapoor Vali, and Kris Wolff. Fred Marotto's book *Introduction to Mathematical Modeling Using Discrete Dynamical Systems*, published by Cengage in 2005, was also an inspiration for this work, though

I did not have the opportunity to discuss our areas of common interest very often. Jack Kamis, Mila Martynovsky and Michael "Misha" Zigelbaum provided me with valuable and practical teaching advice.

In particular, Shapoor Vali and I wrote our books at roughly the same time, so we endured the struggle of writing book chapters, proofreading drafts, and drawing up book-proposals simultaneously. We have both crossed a great desert, not without pain and suffering, and now we can both enjoy the oasis at the other side.

Many Fordham undergraduates gave me a great deal of tips, corrections, opportunities for improvement, and ideas. At the risk of accidentally excluding someone, I wish to mention the following Fordham students: James Compagnoni, Gray Crenshaw, Dan DiPasquale, Patrick Fox, Alex Golec, Simon Kaluza, Kyle Kloster, Peter Muller, Luigi Patruno, Seena Vali, and Sean Woodward. Much encouragement was also given by Ed Casimiro, for which I am grateful.

After the four-year position at Fordham had elapsed, I had the great luck to be hired at the University of Wisconsin—Stout, the polytechnic of the University of Wisconsin system. This was a great boon for me as I received my undergraduate degree in Computer & Systems Engineering from Rensselaer Polytechnic Institute in 1999, and similarly my father received his undergraduate degree in Mechanical Engineering from ETH in Zürich (the polytechnic of Switzerland) back in 1951. Accordingly returning to the atmosphere of a polytechnic was somewhat like coming home. While the three institutions are far from being identical triplets, the noble goal of preparing hard-working students to become engineers, scientists, and managers, or members of other lucrative technical professions, creates an attitude of focus on the pragmatic, on hard work, and on exactitude. This "polytechnic attitude" is reflected across the spectrum of academic rank, from the Chancellor to the freshmen. It breeds respect for mathematics in general, and calculus in particular. At UW Stout, mathematics is seen as a bridge to important career-related coursework, and not as a barrier to graduation, as it was often viewed at Fordham, by both students and high-ranking administrators.

The vast, vast majority of this book was written while working for UW Stout. Both Dean Jeffery Anderson and Dean Charles Bomar provided administrative, financial, and emotional support for this project. The degree of cooperation in our *Department of Mathematics, Statistics, and Computer Science* is extremely rare and heartwarming. Unless I am very much mistaken, every last member of the department has contributed to this book via a suggestion, correction, comment, filling in for me in class when I was away at a conference, or by giving encouragement. Teaching 24 credits per year in a climate thoroughly unfriendly[1] to human habitation, while

[1]This is not an exaggeration. Two examples will have to suffice. On the first day of classes in 2014, namely January 28th, as I was leaving the house to go to campus, `weather.com` gave a temperature of -22°F on the thermometer, and -41°F with the

attempting to maintain some fraction of a research life, has brought us together. With great respect, I inscribe their names here at this time: Anne Antonippillai, Wan Bae, Alexander Basyrov, Chairman Christopher Bendel, Seth Berrier, Jeffrey Boerner, Diane Christie, Steven Deckelman, Brent Dingle, Seth Dutter, Jeanne Foley, Eugen "Andre" Ghenciu, Petre "Nelu" Ghenciu, Nasser Hadidi, Matt Horak, Ayub Hossain, Amitava Karmaker, Deborah Kruschwitz-List, Terrence Mason, Valentina "Diana" Postelnicu, Laura Schmidt, Loretta Thielman, Peter Thielman, Keith Wojciechowski, and Mingshen Wu. Additionally, the former faculty members Joy Becker, John Hunt, Benjamin Jones, Dennis Mikkelson, and Eileen Zito, have provided advice, encouragement, or course materials. Furthermore, several of our temporary hires have voluntarily taken upon themselves either phenomenally unpleasant courses or the elevated obligations of full-time faculty, without additional financial compensation, and are therefore particularly praiseworthy: Jim Church, Brian Knaeble, Shing Lee, Olga Lopukhova, John Neiderhauser, Mark Pedersen, and Dennis Schmidt. The chairman, Chris Bendel, is particularly dedicated and an outstanding peacemaker who has created a work environment of collaboration, cooperation, and mutual respect often dreamed of, but almost never actually found, in academic life. In MSCS, we labor to assist one another, in contrast to another department in our building which I will not explicitly name, where the senior faculty have made a hobby of smashing the careers and shattering the dreams of their own junior faculty.

Outside of my department at UW Stout, Alan Block, Mark Fenton, Julie Furst-Bowe, Sue Foxwell, Jennifer Grant, Mary Hopkins-Best, Michael Levy, Steve Nold, Marlann Patterson, and Todd Zimmermann have given me bountiful encouragement. I would also like to thank the entire Department of Biology who have set a very high standard of scholarship, of interactive teaching, of skilled grant-writing, of innovative and laboratory-focused coursework, and who have taught us all to get undergraduates heavily and materially involved in research.

It would be customary to list here the students at UW Stout who have enriched my teaching and given me useful feedback. Such a list would seriously lengthen the book, however. I have been very lucky, in that I have been given highly applied courses, with content extremely applicable to professional practice in both engineering and management. You will see this effect in the body of this book, as many examples will be drawn from physics and economics in concordance with the professional aspirations of my students. The vast majority of my students at some time or another have given me a suggestion, a correction, or shown me some new application of mathematics to their chosen fields of study. I have studied and taught at other institutions, but this phenomenon is one that I have only encountered at

---

windchill. For my metric readers, that is -29.4°C on the thermometer, and -40.6°C with the windchill. On May 2nd and May 3rd, 2013, we had 17 inches of snow (that's 43.2 cm).

UW Stout, and at Oxford in the UK when I was a visiting student there. Since I am unable to list all the students by name, I will thank them by course number: Math-123, *Finite and Financial Mathematics*; Math-154, *Calculus II*; Math-380/580, *Cryptography*; Math-447/747, *Numerical Analysis II*; and CS-480/680, *Computer Security*.

Previous to teaching at Fordham I had the opportunity to adjunct at American University in Washington DC. I would like to thank Michael Gray, Angela Wu, Nate Harshman, and Michael Black for their support, encouragement, assistance, and mentoring, as well as the alumnus Alexander Ivanov, for his encouragement and personal energy.

In addition to the faculty at universities where I have been teaching, the faculty of Columbia University and of the City University of New York (CUNY), two of the best institutions in "the city" and the nation, have given me great help. I have enjoyed conversations on both research and teaching with David Allen, David Bayer, Maria Chudnovsky, Kenneth Kramer, Hunter Johnson, and Vladimir Shpilrain.

In physics and in chemistry, I have had the tremendous benefit of consulting experts. For physics, it was Gabriel Hannah, and for chemistry, it was Gergely Sirokman. They have checked their relevant sections carefully, and therefore the reader can be confident that those sections are free of mistakes, with one exception. In the physics problems, I have not carried the units through the formulas and into equations, as is required by that subject. In applied mathematics, we generally do not do this, and it would be highly disorienting and tedious for me to modify this already completed book to match the convention of physicists.

I have received a great deal of encouragement in this project, including detailed suggestions, from those listed above, as well as Martin Brock, Bruce Cohen, Carmi Gressel, Marshall Hampton, Benjamin Jones, David Joyner, Craig Larson, Andrey Novoseltsev, Clement Pernet, and Susan Schmoyer.

My former dissertation advisor and trusted friend Lawrence Washington has inspired me with his extremely successful textbooks. He has set a very high bar for the rest of us to aspire to. I frequently recommend *Cryptography with Coding Theory*, by Wade Trappe and Lawrence Washington, to anyone who wants to learn cryptography—from the undergraduate sophomore to the senior faculty member.

The WeBWorK community has worked closely with the Sage community. WeBWorK is the free open-source competitor to the online homework systems run by textbook publishing houses, and is endorsed by the Mathematical Association of America. Numerous hours of the lives of the WeBWorK volunteers have been devoted to writing and maintaining the WeBWorK system, as well as coding math problems. For the integration with Sage, the community is heavily indebted to John Travis, of Mississippi College, who actively contributes to both communities.

The funding for Sage comes in large part from the National Science Foundation, and in particular the divisions listed below. The Sage community is grateful for the support of the National Science Foundation grants:

- DMS-0545904, Division of Mathematical Sciences, "CAREER: Cohomological Methods in Algebraic Geometry and Number Theory,"
- DMS-0555776, Division of Mathematical Sciences, "Explicit Approaches to Modular Forms and Modular Abelian Varieties,"
- DMS-0713225, Division of Mathematical Sciences, "SAGE: Software for Algebra and Geometry Experimentation,"
- DMS-0757627, Division of Mathematical Sciences, "FRG: L–functions and Modular Forms,"
- DMS-0821725, Division of Mathematical Sciences, " SCREMS: The Computational Frontiers of Number Theory, Representation Theory, and Mathematical Physics,"
- DMS-0838212, Division of Mathematical Sciences, "EMSW21–RTG: Research Training Group on Inverse Problems and Partial Differential Equations,"
- DMS-1015114, Division of Mathematical Sciences, "Sage: Unifying Mathematical Software for Scientists, Engineers, and Mathematicians,"
- DUE-1022574, Division of Undergraduate Education, "Collaborative Research: UTMOST: Undergraduate Teaching in Mathematics with Open Software and Textbooks,"
- ACI-1147463, Division of Advanced CyberInfrastructure, "Collaborative Research: SI2-SSE: Sage-Combinat: Developing and Sharing Open Source Software for Algebraic Combinatorics,"

as well as funding from Microsoft, Sun Microsystems, Enthought Inc., and Google.

As I mentioned earlier, the Sage community is enormous, with hundreds of developers in many countries of several continents. Despite the above mentioned financial support, almost all of the work is performed on an entirely voluntary and unpaid basis. For the sacrifice of free time on the part of so many people, the entire Sage community is grateful in the extreme.

Chapter 1

Welcome to Sage!

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

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

1.1. Using Sage as a Calculator

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

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

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

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

Notice that the symbol for addition is the plus sign, and the symbol for subtraction is the hyphen. The symbol for multiplication is the asterisk, and the symbol for division is the forward slash (that's the one found with the question mark, not the backslash.) These are the same rules used by MS-Excel and many computer languages, including C, C++, Perl, Python, and Java.

For compound interest, we'd need to take exponents. The symbol for exponents is the caret, found above the number six on most keyboards. It looks like this "^".

Let's consider 12% compounded annually on a signature loan for 3 years, and a principal of \$ 11,000. We all know the formula to be $A = P(1 + i)^n$, so we would just type in

```
11000*(1+0.12)^3
```

and click "evaluate". You will learn that the answer is 15454.2080000000, or $ 15,454.21 after rounding to the nearest penny. By the way, instead of clicking "evaluate," you can just press shift-enter on the keyboard, and it will do the same thing.

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

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

```
11000*(1+0.12)**3
```

which is actually a throw-back to the historical programming language `FORTRAN` which was most popular[1] during the 1970s and 1980s.

What if you make a mistake? Let's say that I really meant to say $ 13,000 and not $ 11,000. Click on the mistake, and correct it using the keyboard, and change the 11 into 13. Now click "evaluate" again, and all is well.

Grouping Symbols:

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

For example, let's say you need to evaluate

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

So you should not type

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

but rather

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

where the brackets have become parentheses.

Some very old math books use braces { and } as a sort of auxiliary also to the parentheses and brackets. These too, if they are for grouping in a formula, must become parentheses. As it turns out, Sage, as well as modern mathematical books, use the braces { and } to denote sets.

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

---

[1] An irrelevant historical aside: Because rewriting software is a painful and time-consuming process, many important pieces of scientific software remained written in `FORTRAN` until a few years ago, and likewise the same is true of business software in `COBOL`. Each new programming language incorporates features of the previous generation of languages, to make it easier to learn, and accordingly one sees some truly ancient seeds of dead languages once in a while.

Three Mistakes that are 90+% of My Errors in Sage:

There are three mistakes that I make a lot when using Sage. For example, if you want to say $11,000x + 1200$ then you have to type

```
11000*x+1200
```

The first error that I sometimes make is that I leave out the asterisk between `11000` and `x`. In Sage, you must include that asterisk, as that is the symbol of multiplication. The second error that I'll often add a comma inside the `11000`, but it is not acceptable in Sage to write `11,000` for `11000`. The third error is that I'll have mismatched parentheses. Any Sage expression should have the same number of `(`s as it has of `)`s—no more and no less.

A more complete list is given as Appendix A, "What to Do When Frustrated," on Page 295.

1.2. Using Sage with Common Functions

Now I'll discuss how Sage works with square roots, logarithms, exponentials, and so forth.

Square Roots:

The standard "high school" functions are also built-in. For example, type

```
sqrt(144)
```

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

```
sqrt(8)
```

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

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

and obtain

```
2.82842712475.
```

which is a decimal approximation.

The `N()` function, which can also be written `n()`, will convert what is inside the parentheses to a real number, if possible. Usually that is a decimal expansion, thus unless what is inside the parentheses is an integer[2] then it will be, necessarily, an approximation. Sage will assume that all decimals are mere approximations, so for example

[2]To be mathematically correct, I should say "an integer or a fraction with denominator writable as a product of a power of 5 and a power of 2." For example, 25ths, 16ths, and 80ths can be written exactly with decimals, where as 3rds, 15ths, and 14th cannot. Observe that $25 = 5^2$ and $16 = 2^4$ as well as $80 = 2^4 \times 5$. Those denominators have only 2s and 5s in their prime factorization. Meanwhile, $3 = 3$ and $15 = 5 \times 3$ while $14 = 7 \times 2$. As you can see, those denominators have primes other than 2 and 5 in their prime factorization. If you find this interesting, you should read "Using Sage to work with Integers" on Page 145.

```
sqrt(3.4)
```

will evaluate to `1.84390889145858`, without need of using `N()`.

Higher Order Roots:

Higher order roots can be calculated like exponents are. For example, to find the sixth root of 64, do

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

and obtain that the answer is 2.

The Special Constants π and e:

Frequently in math we need the special constants π and e. It turns out that π is built into Sage as "pi" (both letters lower case) and e is built in as "e" (again, lower case).

You can do things like

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

to get many digits of pi, or likewise

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

to get a high-accuracy expansion of $\sqrt{2}$. In this case, we get 200 digits. You can also abbreviate with

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

as we did earlier. That's what the `N()` or `n()` function (they are the same) is an abbreviation of, namely "numerical approx."

Tab Completion:

Now would be a good time to explain a neat feature of Sage. If you are typing a long command, like "`numerical_approx`" and part way through you forget the exact ending (is it approximation? approximations? appr?) then you can hit the tab button. If only one command in the entire library has that prefix, then Sage will fill in the rest for you. If it does not, then you get a list of suggestions. It is a very useful feature when you cannot remember exact commands. We will see other examples of built-in help features in Section 1.10 on Page 47.

There's another keyboard shortcut that I like. While I mentioned it before, if you are tired of clicking "evaluate" all the time, you can just press Shift and Enter at the same time. This does the same thing as clicking "Evaluate."

Is Sage Case-Sensitive?

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

The only four exceptions known to me are i and `n()`, as well as `True` and `False`. We will not make extensive use of `True` and `False` until Chapter 5, but it is harmless to mention that you can also enter them as `true` and `false`, and Sage recognizes these as the same. You may have heard that $\sqrt{-1}$ is often referred to as "the imaginary number" and is denoted i.

Next, you can represent $\sqrt{-1}$ as either `i` or `I` and either way Sage will know what you meant, namely $\sqrt{-1}$. If you've never heard of imaginary numbers or i, then you can safely ignore this fact. Lastly, you can write `n(sqrt(2))` or `N(sqrt(2))` and Sage treats those as identical.

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

Exponentials:

Just as

```
2^3
```

gives you 2^3 and likewise

```
3^3
```

gives you 3^3, if you want to say e^3 then just say

```
e^3
```

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

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

Also, it is worth mentioning sometimes books will write

$$\exp(5 \cdot 11 + 8) \text{ instead of } e^{5 \cdot 11 + 8}$$

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

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

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

Logarithms:

Of course, Sage knows about logarithms—but there's a problem. There are several types of logarithm, including the common logarithm, the natural logarithm, the binary logarithm, and the logarithm to any other base you feel like using. In high school "log" refers to the common logarithm, and "ln" to the natural logarithm. However, after calculus "log" refers to the natural logarithm. Since Sage was mainly meant for university-and-higher level work, then it is only natural they chose to use the natural logarithm for "log."

So to find the natural logarithm of 100 type

```
N( log(100) )
```

for the common logarithm of 100 type

```
N( log(100,10) )
```

and for the binary logarithm of 100 type

```
N( log(100,2) )
```

which of course generalizes. For example, to find the logarithm of 100 taken base 42, type

```
N( log(100,42) )
```

Note that Sage is quite good at getting exact answers. Try

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

and you will obtain 3, the exact answer.

A Financial Example:

Suppose you deposit $ 5000 in an account that earns 4.5% compounded monthly. Perhaps you are curious when your total will reach $ 7000. Using the formula $A = P(1+i)^n$, where P is the principal, A is the amount at the end, i is the interest rate per month, and n is the number of compounding periods (number of months), we have

$$\begin{aligned} A &= P(1+i)^n \\ 7000 &= 5000(1+0.045/12)^n \\ 7000/5000 &= 5000(1+0.045/12)^n \\ 1.4 &= (1+0.045/12)^n \\ \log 1.4 &= \log(1+0.045/12)^n \\ \log 1.4 &= n\log(1+0.045/12) \\ \frac{\log 1.4}{\log(1+0.045/12)} &= n \end{aligned}$$

And so, we type into Sage

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

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

This is an example of a computer-assisted solution to a problem. In addition to that approach, Sage is also willing to solve the problem from start to finish, with no human intervention. We'll see that on Page 46.

Pure Mathematics and Square Roots:

Here are a couple of notes about square roots that would appeal to math majors, or anyone who wants to work with the complex numbers. Most students will want to skip this subsection, as well as the next one on complex numbers, and continue with either "Using Sage for Trigonometry" on Page 7, or with "Using Sage to Graph 2-Dimensionally" on Page 8.

In the spirit of absolute pomposity, you may know that $(-2)^2 = 4$ as well as $2^2 = 4$. So if you want *all* the square roots of a number you can do

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

to obtain

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

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

A Taste of the Complex Numbers:

If you know about complex numbers, you can do

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

to obtain

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

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

While Sage is quite good at complex analysis, and can produce some rather lovely plots of functions of a complex variable, we will not go into those details here. The color plots of complex-value functions will be covered in the electronic-only online appendix to this book "Plotting in Color, in 3D, and Animations," available on my webpage `www.gregorybard.com` for downloading.

1.3. Using Sage for Trigonometry

Some students are unfamiliar with trigonometry, or are in a course that has nothing to do with trigonometry. If so, then they may confidently skip this section and jump to the next one, which begins on Page 8.

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

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

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

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

and obtain `1/12*(sqrt(3) + 3)*sqrt(6)` which is especially unsatisfying if you want a decimal. Instead, if you type

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

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

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

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

will produce the helpful answer "Infinity." The "rare" or "reciprocal" trigonometric functions of cotangent, secant, and cosecant, which are important

in calculus but annoying on hand-held calculators, are built into Sage. They are identified as `cot`, `sec`, and `csc`.

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

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

you will obtain

```
1/6*pi
```

as expected. Likewise

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

produces

```
1/3*pi
```

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

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

You can also use Sage to graph the trigonometric functions. We'll do that in the section entitled "Using Sage to Graph 2-Dimensionally," on Page 11.

Converting between Degrees and Radians:

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

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

produces `60` while

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

produces `1/3*pi`.

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

1.4. Using Sage to Graph 2-Dimensionally

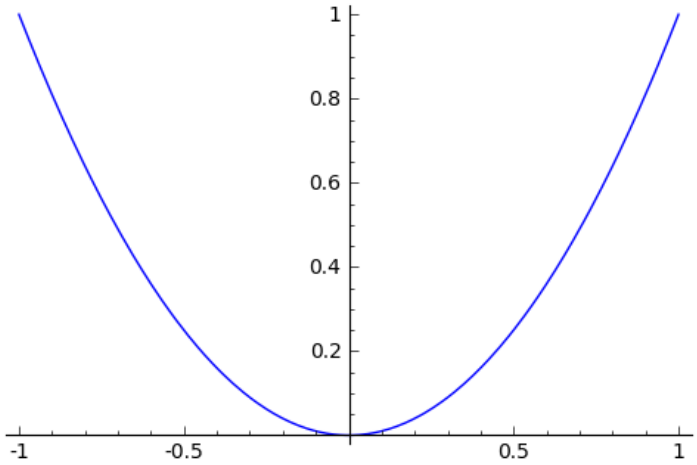
My favorite shape is the parabola, so let's start there. Type

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

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

$$-1 < x < 1$$

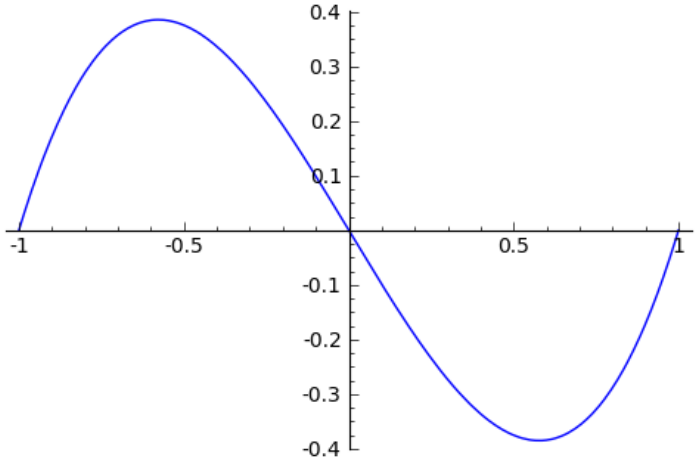
which is the default range for the `plot` command. It will bound the y-values to whatever is needed to show those x-values. Here's a screenshot of what you should see:



Likewise you can do

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

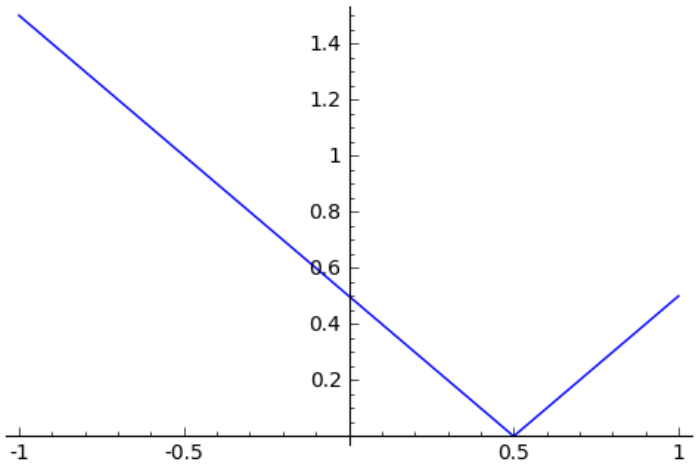
which is nice and visually appealing, as you can see:



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

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

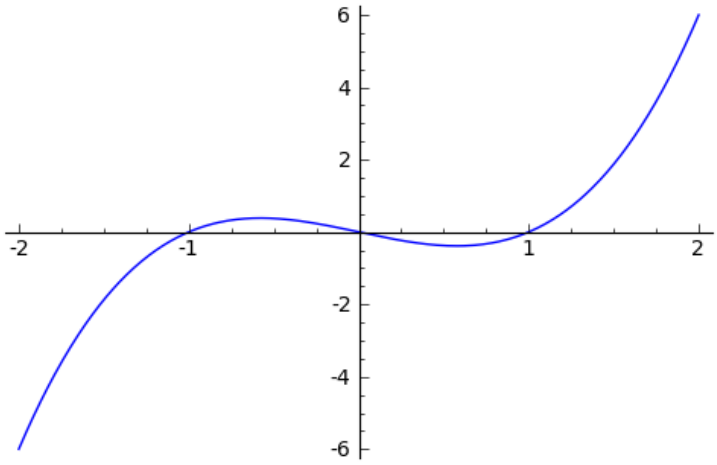
which produces the following screenshot



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

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

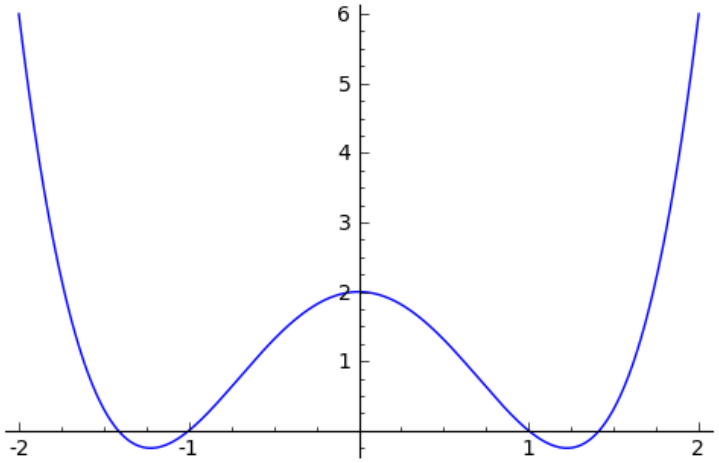
and you get the desired graph, namely:



Now would be a good time to mention that you can save any of these plots to a file after having Sage generate them. Just right-click on the displayed graph in your web browser, and save the file to your computer. You can then import the image into any report or paper that you might be writing. A very cool graph is

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

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



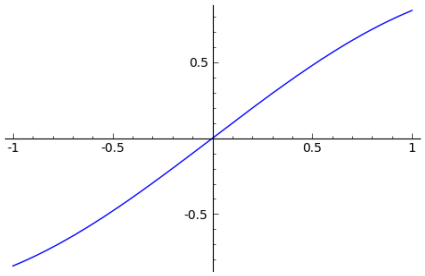
Another minor sticking point is that for $y = \sin(t)$ you have to say

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

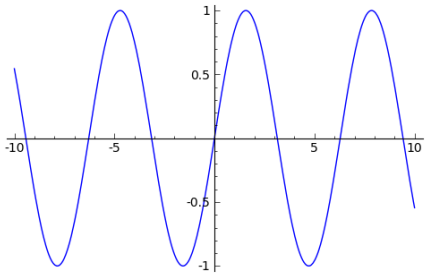
or better yet

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

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



`plot(sin(x))`



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

In fact if you were to type

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

you would see

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

because in this case, Sage does not know what 't' means. An alternative way of handling this is given on Page 104.

1.4.1. Controlling the Viewing Window of a Plot

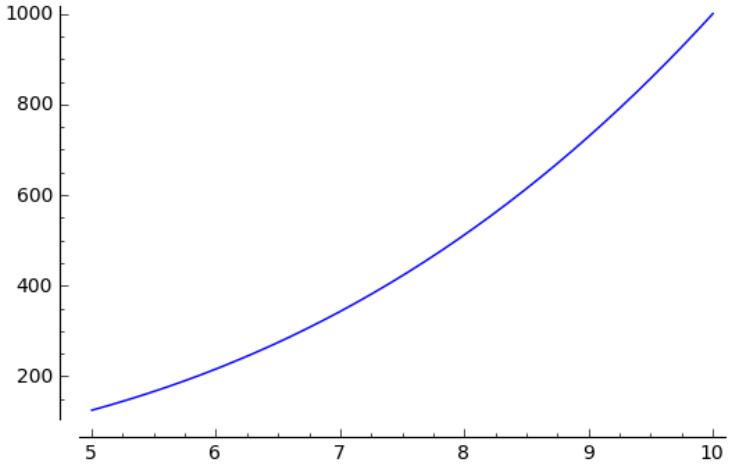
Much of the time, the default viewing window of a graph is going to be exactly what you wanted—but not always. The most common exception is a function with vertical asymptotes. Here we are going to discuss how to adjust the viewing window.

Going "Off the Scale":

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

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

and as you can imagine, the function goes from $5^3 = 125$ up to $10^3 = 1000$, thus the origin should appear very far below the graph. This is the plot:

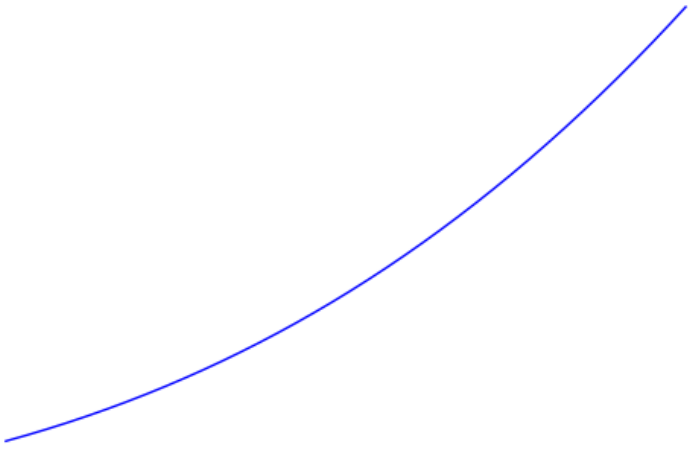


Your hint that the location of the x-axis in the display is not where it would be normally is that the axes do not intersect. This is to tell you that the origin is far away. When the axes do intersect on the screen, then the origin is (both in truth and on the screen) where they intersect.

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

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

and that produces



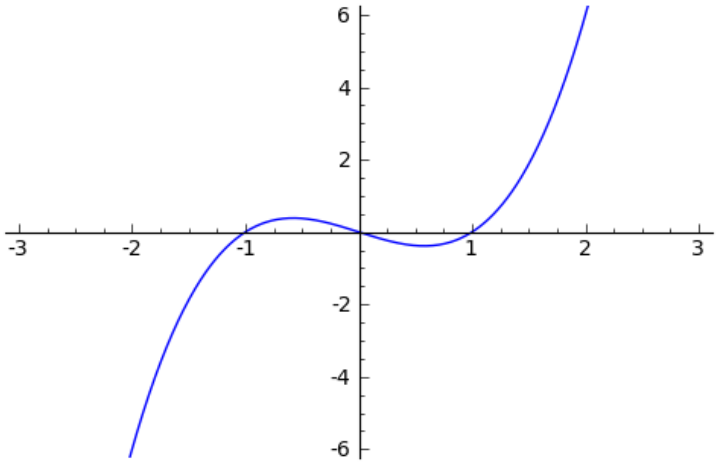
which is considerably less informative.

To Force the Y-Range of a Graph:

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

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

in order to obtain

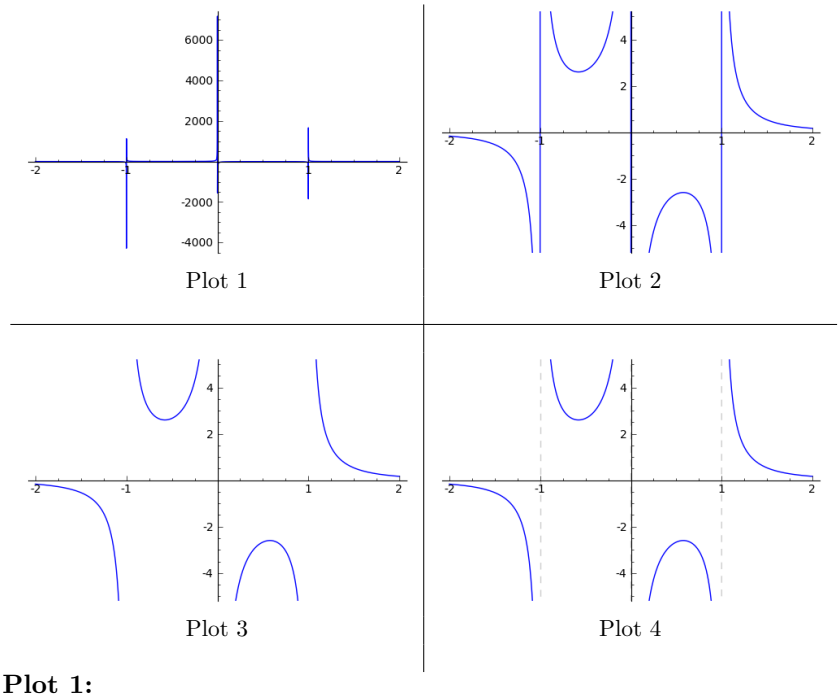


This is important, because normally Sage wants to show you the entire graph. Therefore, it will make sure that the y-axis is tall enough to include every point, from the maximum to the minimum. For some functions, either the maximum or the minimum or both could be huge.

Plots of Functions with Asymptotes:

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

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



Plot 1:

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

Plot 2:

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

Plot 3:

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

Plot 4:

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

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

One minor point remains. Did you notice in the four plots above, that `'show'` is in quotes, but `True` is not in quotes? That's because `True` and `False` occur so often in math, that they are built-in keywords in Sage. However, the `'show'` occurs less often, and therefore is not built in, and we must put it in quotes.

1.4.2. Superimposing Multiple Graphs in One Plot

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

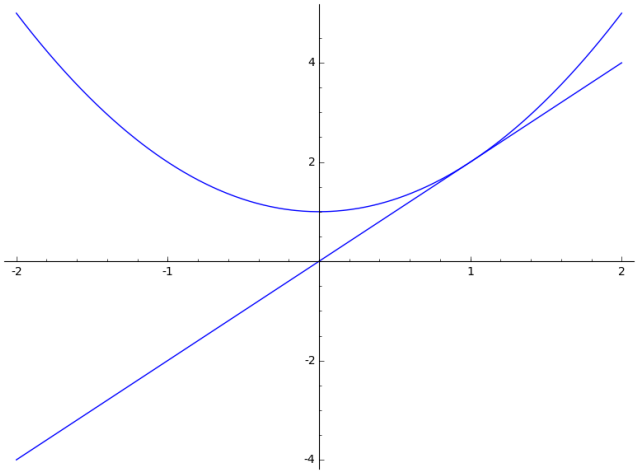
An Example from Calculus: The Tangent Line

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

The command for this will be

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

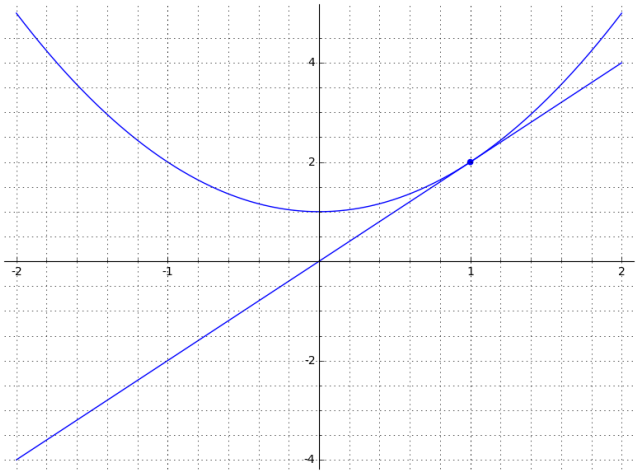
and we will get the image:



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

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

and we will get the image:



As you can see, we added the point using the `point` command and this too was superimposed using the plus sign. The `gridlines='minor'` gave us the very satisfactory grid which can be so useful in preparing nice graphs that are readable, for lab reports, classroom use, or papers for publication. We will discuss other ways of making gridlines, as well as other ways of annotating a graph, in Section 3.1 on Page 91. This specific graph will appear in a highly annotated way on Page 95.

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

A More Intricate Example:

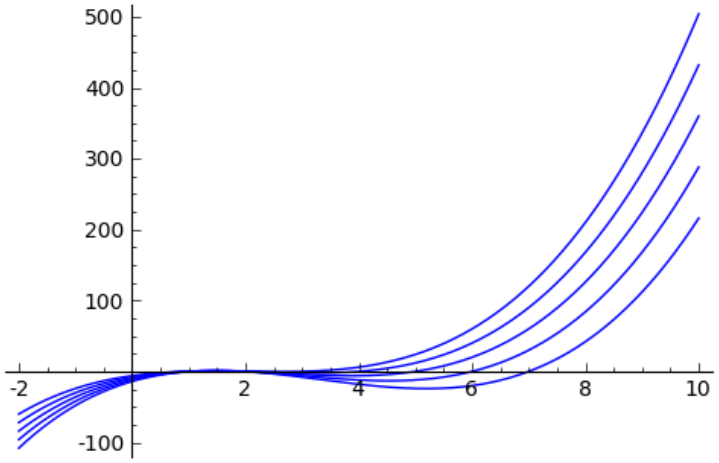
Suppose you were investigating polynomials, and you wanted to know the effect of varying the last numeral in

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

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

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

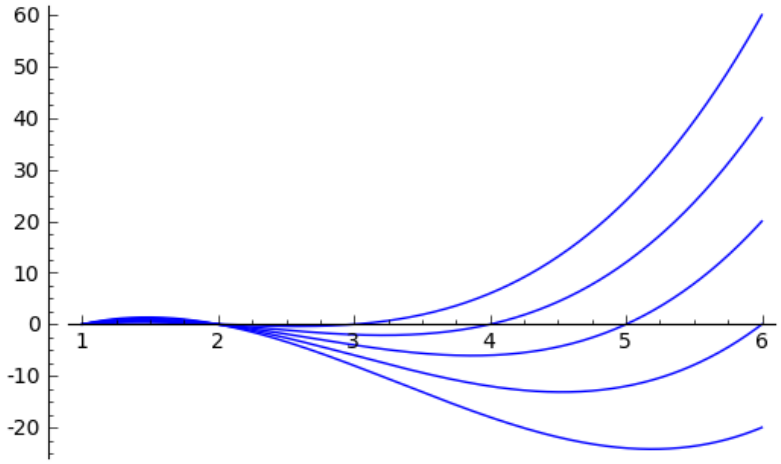
where I chose the domain, $x = -2$ until $x = 10$, arbitrarily. By the way, when you enter that previous command in, it is important to note that it must be on one (very long) line. If done correctly, we obtain



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

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

and obtain as a result



There is an important issue here. We have to be careful to make the changes correctly, and that somewhat challenging requirement is made much worse because the code is hard to read. However, we will soon learn how to make the code more readable, and this will also make the code easier to change.

In any case, now I can see better what that last numeral does. It controls the location, or x-coordinate, of the last time that the curve crosses the x-axis. As you can see, the plus sign among the `plot` commands means to superimpose them.

There is a technicality here. You must not break a newline among the plots and plus signs. Therefore, if you have problems getting those last two commands to work, then check to see that you have no accidental "line breaks" in the command. For Sage, these commands must be entered without a line break, kind of like typing a long paragraph in an ordinary word processor—you can wrap around the end of the line, but you cannot insert a line break. This is an example of code that is functional, but not easy to read, nor change, and certainly not easy to understand. We can fix this, with a bit of effort. The various plot commands and their plus signs must be strung together, like a paragraph, wrapping naturally from one line to the next. You do not use the enter key to put each plot on its own line. Thus, you cannot do:

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

which Sage thinks is five separate commands, instead of one big command.

However, there is a nice way out of this dilemma. You can type the following, which produces the desired graph.

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

P = P1 + P2 + P3 + P4 + P5
P.show()
```

I'm sure we can all agree that this is much more readable code.

Further Graphing & Plotting Topics:

We have now only scratched the surface of plotting in Sage. There is a rich library of types of plots and graphs which Sage can produce for you. An entire chapter of this book is dedicated to "Advanced Plotting Techniques" beginning on Page 91. If you're curious about how Sage actually generates the images, we talk about that on Page 267, but you might have to read most of Chapter 5 to be able to follow that discussion.

By the way, you do not need to read that entire chapter in order. You can simply use the index or table of contents to select exactly which type

of graph or plot you would like, and just read that tiny subsection of that chapter.

1.5. Matrices and Sage, Part One

In this section we'll learn how to solve linear systems of equations using matrices in Sage. This section assumes that you've never worked with matrices before in any way, or alternatively, that you have forgotten them. Experts in linear algebra can skip to the Section 1.5.3 on Page 22. On the other hand, some readers will have no specific interest in matrices, and can therefore skip to Section 1.6 on Page 30, where they will learn how to define their own functions in Sage.

1.5.1. A First Taste of Matrices

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

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

First you would convert those equations into the following matrix:

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

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

Matrices have a special form called "Reduced Row Echelon Form" often abbreviated as RREF. The RREF is, from a certain perspective, the simplest description of the system of equations that is still true. The Reduced Row Echelon Form of this matrix A is

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

We can translate this literally as

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

or in simpler notation

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

which is the solution. You should take a moment now, plug in these three values, and see that it comes out correct.

1.5.2. Complications in Converting Linear Systems

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

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

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

Consider this system of equations:

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

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

- In the first equation, the variables are out of order, which is extremely common. Very often, both students and professional mathematicians will fail to notice this, and put the wrong coefficients in the wrong places. You can use any ordering that you want, so long as you use the same ordering for every equation. However, to avoid making an error, mathematicians usually put the variables in alphabetical order.

- Also in the first equation, the w is on the wrong side, and we must move it to the left of the equal sign. It is necessary that all the variables end up on one side of the equal sign, and all the numbers on the other side of the equal sign. With these two repairs, the first equation is now

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

- The second equation has the constant on the wrong side, and so we must move it across the equal sign, remembering to negate it.
- As if that were not enough, both x and w are missing in the second equation. We treat this as if the coefficients were zero. With these two repairs, the second equation is now

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

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

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

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

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

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

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

The RREF of that matrix would be tedious to find by hand. However, it would be far worse to solve that system of equations by hand without matrices, using ordinary algebra. However, using Sage, we will easily compute the RREF in the next subsection. For now, it turns out that the answer is

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

which you can easily verify now if you like.

1.5.3. Doing the RREF in Sage

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

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

which came from the previous subsection. As you can see, it has 3 rows and 4 columns. We'll do that with the Sage command

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

The key to understanding that Sage command is to see that the first number is the number of rows, followed by the number[3] of columns. Then follows the list of entries, separated by commas, and enclosed in brackets.

By the way, keep this definition of the matrix A in the Sage Cell for as long as you are working with it. Do not erase it until you are done with it. Every time you hit evaluate, you are sending whatever is in the Sage Cell to the Sage Cell Server for evaluation. The server won't remember the definition of the matrix A if you remove it. In contrast, SageMathCloud (See Appendix B), has much more permanent memory.

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

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

The third step is that you want to compute the RREF or "Reduced Row Echelon Form." This is done with the command

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

and Sage tells you the Reduced Row Echelon Form, or

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

The fourth, and optional step, is to check your work, but I highly recommend it! First we type (on its own line)

```
print 3*3 + -4*0 + 5*1
```

and learn that the answer is 14, as expected. Next we type

[3]This is how I remember: Columns, such as on a bank, Congress, or a Greek temple, are vertical. Rows, by process of elimination, are horizontal. But do we enter the rows first, or the columns first? I always think "RC Cola," a minor brand of soda, which reminds me rows-first-columns-second. Thankfully, there does not exist a "CR Cola."

```
print 1*3 + 1*0 - 8*1
```

and learn that the answer -5, as expected. The last equation is checked similarly.

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

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

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

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

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

and we can enter that into Sage with

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

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

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

In Sage, the naming of the coordinates in matrix entries starts at 0 and not at 1. This means that the upper-left hand corner is `B[0][0]` in Sage, but B_{11} in mathematics. Similarly, B_{14} is typed in as `B[0][3]`, and B_{23} is typed in as `B[1][2]`. This anomaly is something that Sage inherited from the computer language Python, and it cannot be changed since Sage runs over Python.

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

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

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

columns. You can feel free to use which ever format you might happen to feel more comfortable with.

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

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

which gives the final answer of

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

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

```
print 2*(-12/7) - 5*(19/7) + 54/7
```

and

```
print 6 + -107/7
```

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

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

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

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

If you happened to read Section 1.5.2, you will recall that we did quite a lot of work to get B. Those many steps might have contained an error if we were sloppy. That's another reason that we should check our work.

1.5.4. Basic Summary

We've now learned how to use the `matrix` and `rref` commands to compute the RREF of a matrix, and therefore solve a linear system of equations. To do all the steps at once, I will often type

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

print "Question:"
print B
print "Answer:"
print B.rref()
```

The lines containing `print "Question:"` and `print "Answer:"` are entirely optional, but help make the output more readable. They also encourage me to check that I had actually entered the matrix which I had intended

to enter, without any typos. Somehow, I almost always make a typographical error, which I have to go back and correct. The output produced is below:

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

1.5.5. The Identity Matrix

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

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

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

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

1.5.6. A Challenge to Practice by Yourself

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

$$\begin{aligned} 3481x + 59y + z &= 0.87 \\ 6241x + 79y + z &= 0.61 \\ 9801x + 99y + z &= 0.42 \end{aligned}$$

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

1.5.7. Vandermonde's Matrix

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

Sometimes it would be useful to know the curve but it can usually only be computed by surveying potential customers. The standard technique is to find 2000 people, show each of them the product, and ask them if they would buy it or not. They get the product[4] for free in return for participating in the survey. The key is that of those 2000 people, perhaps 20 different prices would be used, each price being shown to 100 people.

That process will deliver 20 data points, which is often totally sufficient. However, sometimes it is better to have an actual function. Typically, a quadratic function is sought. There are good reasons for this, but it would be a digression[5] to explore them here. Imagine that a friend of yours with a startup company has done such a survey, but with 3 prices and 300 people, for some snazzy new product.

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

Every quadratic function can be written in the form

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

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

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

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

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

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

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

matrix down:

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

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

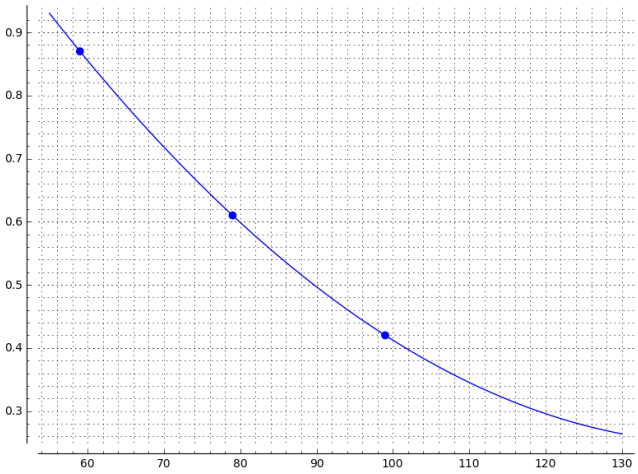
$$a = 0.0000875 \qquad b = -0.025075 \qquad c = 2.0448375$$

as practice, on Page 25 as Section 1.5.6.

Finally, we conclude that the function is

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

The graph of this function is given below:



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

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

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

The general technique of fitting a degree-n polynomial to $n+1$ points using a matrix is due to Alexandre-Théophile Vandermonde (1735–1796), and therefore the matrices are called Vandermonde Matrices after him. For example, to fit a fourth-degree polynomial to five data points (x_1, y_1), (x_2, y_2), $\ldots$, (x_5, y_5), we would use the matrix:

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

1.5.8. The Semi-Rare Cases

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

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

That results in the matrix

$$C1 = \left[\begin{array}{ccc|c} 1 & 2 & 3 & 7 \\ 4 & 5 & 6 & 16 \\ 7 & 8 & 9 & 24 \end{array}\right]$$

and that we shall enter into Sage with

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

Next, to find the RREF we type

```
C1.rref( )
```

and receive back

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

This translates into

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

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

Another interesting case is to change the 24 in the last equation to 25. That results in the matrix

$$C2 = \left[\begin{array}{ccc|c} 1 & 2 & 3 & 7 \\ 4 & 5 & 6 & 16 \\ 7 & 8 & 9 & 25 \end{array}\right]$$

and that we shall enter into Sage with

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

Now, to find the "solution," we type

```
C2.rref( )
```

and receive back

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

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

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

When are there Infinitely Many Solutions?

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

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

However, be warned! This rule does not work if there are "too few" or "too many" equations. If there are too many equations (too many rows), the system is said to be "over defined." If there are too many variables (too many columns), the system is said to be "under defined." For example, in Section D.3 on Page 314, I will list some counterexamples that demonstrate that this rule of thumb is false for matrices that are over defined or under defined. If you are curious, under-defined systems of linear equations come up in some cool analytic geometry questions, and over-defined systems of linear equations come up in cryptanalysis—and were used in my PhD dissertation.

Finally, it should be noted that in Appendix D, you can learn the relatively straight-forward technique that maps out exactly how to handle linear systems of equations with infinitely many solutions, regardless of the number of rows and columns. The trick there is only slightly more complicated, but it always works. Furthermore, it gives you formulas for producing as many solutions from the set of infinitely many solutions, as you might want to generate. You can also use those formulas for other purposes—for example, to test if a candidate solution is actually one of the infinitely many solutions.

Please do not imagine that linear systems with infinitely many solutions are terribly rare—they do occasionally come up in applications. However, they are uncommon enough that we will not burden you with those details yet, but we will wait until Appendix D.

1.6. Making your Own Functions in Sage

It is extremely easy to define your own functions[6] in Sage. To me, this is one of the most brilliant design features of Sage, that the creators made functions very easy to use. You can make your own function using exactly the symbols that you would normally use in writing the function down with your pencil on homework or writing on the whiteboard.

Defining a Function:

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

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

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

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

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

To evaluate several values, you can use the `print` command.

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

[6]Just to clarify, I am talking about mathematical functions, such as $f(x) = x^3$. However, if you are an experienced computer programmer, you might know about defining functions in a programming language—sometimes called procedures, methods, or subroutines. That will be discussed in Section 5.2 on Page 226.

```
print g(0.00001)
print g(0.000001)
```

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

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

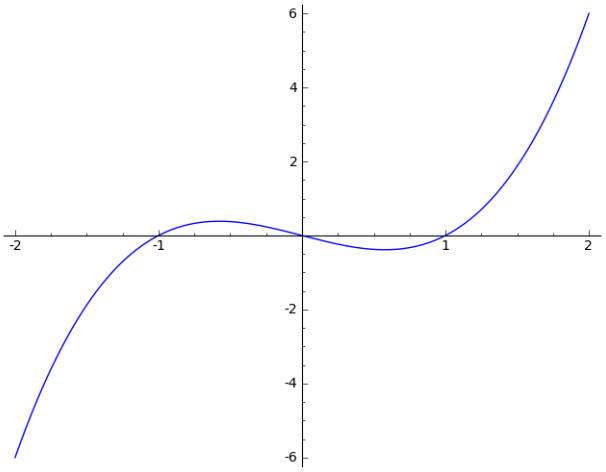
numerically, or help confirm your algebraic answer. If you haven't studied calculus, then you can ignore this remark!

Plotting Functions:

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

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

you get the expected plot



Likewise if you type

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

which is equivalent to

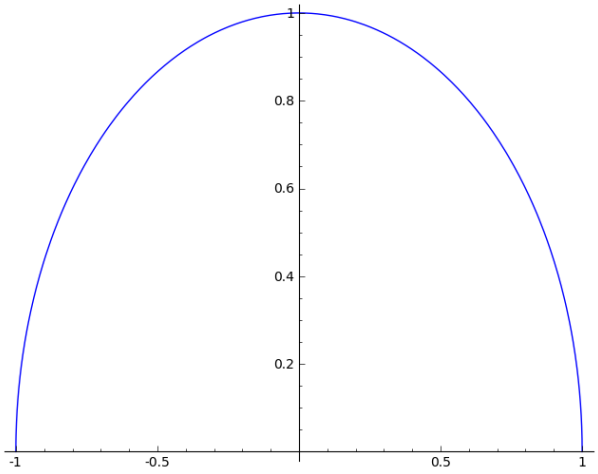
```
g(x) = sqrt( 1-x^2 )
plot( g(x), 0, 1 )
```

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

Last but not least, if you type

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

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



Note that you really do have to keep the definitions of $f(x)$ or $g(x)$ in the window and written out each time you hit "Evaluate." This is a feature of the Sage cell server. It has amnesia, and treats each press of the button "Evaluate" as its own separate mathematical universe. This is for the beginner's benefit, because it makes the correcting of minor typographical errors very easy. You just fix the error, click "Evaluate" and all is well. SageMathCloud works just slightly differently, as will be explained on Page 301.

Using Intermediate Variables:

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

```
k=355/113
```

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

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

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

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

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

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

approximation of π we would remember the formula

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

and type

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

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

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

and learn that this much more common approximation of π as 22/7 has a relative error around 402.5 parts per million. That's much worse! In fact, 22/7 is 4740.9 times worse, so it is kind of a pity that we do not teach students 355/113, but instead we teach them 22/7.

Composition of Functions:

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

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

Perhaps we wish to explore $f(g(x))$. The easy way to do this is to define a third function, $h(x) = f(g(x))$. Then perhaps we might want to print that function, and the values of $h(x)$ at $x = 1$, $x = 2$, and $x = 3$. The Sage code for this is straight forward:

```
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)
```

That produces the output

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

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

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

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

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

which is completely different than

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

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

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

We'll cover this in detail in Section 4.1 on Page 127.

1.7. Using Sage to Manipulate Polynomials

Now we're going to explore some uses of functional notation, from the previous section, to polynomials. First, let's see that Sage can add polynomials very easily. We're going to explore the following polynomials.

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

If we type the following code

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

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

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

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

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

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

but this provides the unfinished yet undeniably true answer

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

Instead, we can do the following

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

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

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

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

The symbol `|-->` means "evaluates to." What Sage is telling you here is that $g(x)$ is a function that sends x to $x^4 - 13x^3 + 61x^2 - 123x + 90$.

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

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

You can also factor more directly with

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

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

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

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

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

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

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

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

It should be noted that these examples were trivial because we used quadratic polynomials. The following polynomial is one that I would not want to factor by hand—though, admittedly, the integer roots theorem[7]

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

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

f(x) = a(b(x))

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

FIGURE 1. Some Sage Code to Explore Composing Two Functions

would produce the correct answer using only a pencil and a great deal of human patience

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

Speaking of the integer roots theorem, we could type `factor(46189)` and get `11 * 13 * 17 * 19` to help us find roots of that quartic polynomial. The `factor` command deals with both factoring a polynomial and factoring an integer. The command `divisors(46189)`, as an alternative, lists all the positive integers which divide 46,189. Naturally, that's a longer list than `factor`, which only would list the prime divisors (and their exponents, if any). Compare `factor(2048)` and `divisors(2048)`, if you like.

The composition of functions is also not a problem. The code given in Figure 1 will compose two polynomials and produce the answer in three forms. It is noteworthy that this is our first real "program" in Sage, in the sense that it is the first time that we've used more than two or three commands at once.

In particular, our program produces the output:

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

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

Just as we saw at the end of Section 1.6, you get a different answer if you change `f(x) = a(b(x))` into `f(x) = b(a(x))`. Go ahead, give it a try, and see for yourself!

1.8. Using Sage to Solve Problems Symbolically

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

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

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

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

The former is an example of a symbolic solution, and the latter a numerical solution. Often, a numerical answer is desired. However, if the answer is a formula, then a symbolic solution is the only way to go. This section is about symbolic solution, and then the next section is about numerical solution.

1.8.1. Solving Single-Variable Formulas

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

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

and then we obtain

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

It is important to remember the asterisk! To be precise, to type

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

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

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

which outputs

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

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

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

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

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

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

which returns `3.14159265350241`. That could easily be mistaken for π. In fact it is really close to π with relative error 2.78×10^{-11}.

There is no restriction to polynomials. You can also type

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

to get

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

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

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

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

Declaring Variables:

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

1.8.2. Solving Multivariable Formulas

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

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

and then we type

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

and receive back

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

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

Of course, you probably know the quadratic formula very well. (At least I hope you do!) However, you probably do not know Cardano's Cubic Formula. In Section 1.8.5 on Page 43, we'll see how Sage can "rediscover" Cardano's formula on the spot when similarly challenged.

Declaring Several Variables at Once:

By the way

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

is merely an abbreviation for

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

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

1.8.3. Linear Systems of Equations

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

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

which would be done by typing

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

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

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

A more typical linear system might be

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

and to solve that we'd type

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

and Sage gives the answer

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

which is correct. Of course, this is exactly the linear system of equations that you would use if someone asked you to find the parabola connecting the points $(3, 32)$ and $(2, 15)$ as well as $(1, 6)$. That would be $f(x) = 4x^2 - 3x + 5$. This is another example of a Vandermonde Matrix (except that we didn't use a matrix here), as we saw defined in Section 1.5.7 on Page 26.

Naturally, linear systems of equations can also be solved with matrices. In fact, Sage is quite useful when working with matrices, and can remove much of the tedium normally associated with matrix algebra. That was discussed in the matrix section (Section 1.5) on Page 19.

1.8.4. Non-Linear Systems of Equations

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

$$x^6 - 21x^5 + 175x^4 - 735x^3 + 1624x^2 - 1764x + 720 = 0$$

which can be done by

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

which gives

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

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

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

and then we can type `print answer[0]` or `print answer[1]` to get the first or second entries. To get the fifth entry of that list, we'd type `print answer[4]`. This is because Sage is built out of Python, and Python numbers its lists from 0 and not from 1. There are reasons for this, based on some very old computer languages.

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

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

we would type

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

which produces

```
[[p== 1, q== 8, x == -4/3*sqrt(10) - 2/3, y ==
1/6*sqrt(2)*sqrt(5) -2/3], [p== 1, q== 8, x == 4/3*sqrt(10)
- 2/3, y == -1/6*sqrt(2)*sqrt(5) - 2/3]]
```

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

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

Now we can type

```
print answer[0]
print answer[1]
```

and we get

```
[p==1, q==8, x== -4/3*sqrt(10) - 2/3, y== 1/6*sqrt(2)*sqrt(5) - 2/3]
[p==1, q==8, x== 4/3*sqrt(10) - 2/3, y== -1/6*sqrt(2)*sqrt(5) - 2/3]
```

This is Sage's way of telling you:

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

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

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

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

which is Sage's way of telling you that it has already given you all the answers for that problem—the list does not contain a 3rd element.

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

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

and obtain

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

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

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

and then type

```
print answer[0]
print answer[1]
print answer[2]
print answer[3]
print answer[4]
```

and that produces four answers and an error message

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

```
IndexError: list index out of range
```

which is Sage's way of telling you

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

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

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

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

but that there is no "Solution 5."

Because finding the roots of a polynomial (or finding the solutions to an equation) is a common mathematical task, there are some other ways to display the solutions. For example, there's some code which will display each root on its own line, and you can choose to show only real roots, only rational roots, or only integer roots. I will describe that in more detail in Section 5.7.10 on Page 275.

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

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

and we receive back

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

both of which satisfy the given equation. Note that `log` in Sage means the natural logarithm, as was explained on 5.

We can also do

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

and obtain

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

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

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

Complex Numbers:

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

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

which produces

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

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

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

and Sage returns the answer -1.

1.8.5. Advanced Cases

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

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

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

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

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

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

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

so we can ask Sage to solve it with

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

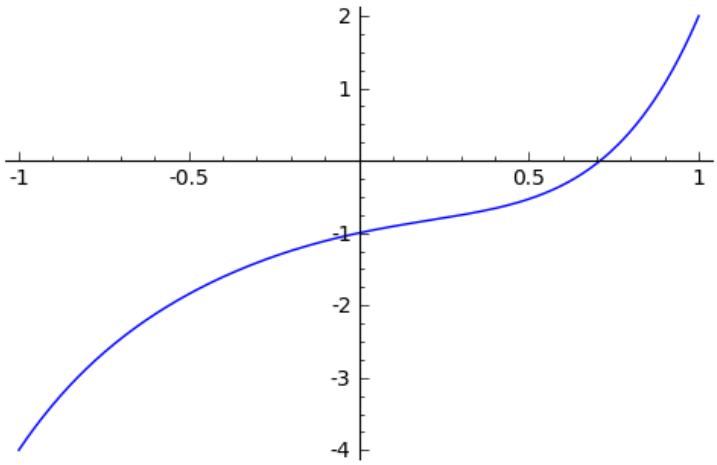
and we get an answer that is a formula of horrific proportions.

1.9. Using Sage as a Numerical Solver

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

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

a polynomial whose graph is given below.



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

To solve for that root, you need only type this

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

and Sage tells you that

$$0.71043425578721398$$

is the root. This is a numerical approximation, because quintic polynomials have a very special property—if you don't know what the special property is, just ask any math teacher.

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

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

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

then Sage tells you

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

which is perfectly honest, because there's no root in the interval for Sage to go and find! If you have no idea where the root is, you can type

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

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

Extremely sophisticated problems can be asked about. Here's a favorite: if someone asks you about $x^x = 5$, then you could try to find where $x^x - 5 = 0$. You would type

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

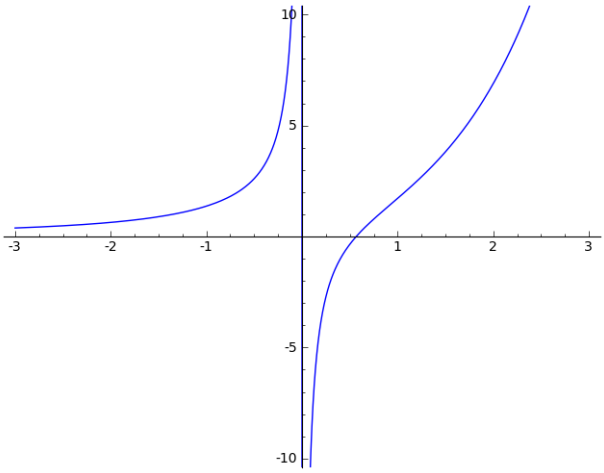
or equivalently

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

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

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

obtaining the plot

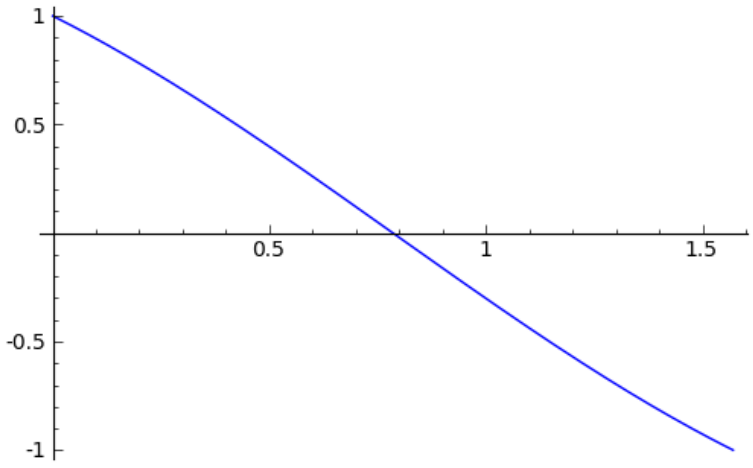


and then you can see the root is between -1 and 1. This means that we should type

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

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

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



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

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

which is just easier notation.

Returning to an Old Problem about Compound Interest:

Earlier, on Page 6, we saw a problem involving compound interest, and finding how many months will be required to turn $ 5000 into $ 7000. Now let's see how to solve that with Sage in only one line, rather than the previous approach which required the human to do some algebra.

The first line of our analysis was

$$7000 = 5000(1 + 0.045/12)^n$$

and translating that into the language of Sage results in

```
7000 == 5000*(1+0.045/12)^n
```

then we should type

```
var('n')
find_root( 7000 == 5000*(1+0.045/12)^n, 0, 10000)
```

Note that the second line means that I'm searching for a value of n that makes the equation satisfied, and that the value of n should be between 0 and 10,000. This seems reasonable, because 10,000 months is a little over 833 years—surely it won't take that long. Note that it is vital to not include a comma inside of 10,000, as Sage's syntax forbids that use of the comma, as we saw on Page 2 and Page 3. The first command above (`var`) is just declaring n as a variable, which we saw on Page 38. In any case, we get the answer `89.894060933080027` as before, and learn that 90 months will be required.

1.10. Getting Help when You Need It

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

The one that I use most commonly is called tab-completion (first described on Page 4), and is meant for when you do not remember the exact name of a command. For example, perhaps I am confused if the correct command is `find_root`, `findroot`, `find_roots`, `root_find`, or maybe something else entirely! I would type `find` and then press the tab key on my keyboard (without clicking "Evaluate.") I would then get a list of choices, most of which have nothing to do with my intentions. However, one is the command that I seek, `find_root`. Alternatively, if I had reached `find_r` before pressing the tab key, then there is only one possible command, and therefore Sage just completes the command for me.

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

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

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

the change for peer review. If the change is found to be helpful and valid, it will be made. Sometimes the entire process can take as little as six weeks.

This now brings up an important point: you have to know the name of the command to use either of those first three methods of getting help. What if you cannot remember the name of the command? This happens often, and there are several solutions.

The official *Sage Reference Manual* can be found at

`http://www.sagemath.org/doc/reference/`

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

Tutorials are much more useful for most readers. Some are organized around a mathematical topic and others are organized around a target audience. There is a collection of Sage tutorials, a partial list of which can be found in Appendix C on Page 305.

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

```
search_doc("laplace")
```

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

```
html/en/constructions/calculus.html:231:<div class="section"
          id="laplace-transforms">
```

The entire line refers to a webpage. The middle number, between the columns, is a line number—one is the top line, two is the second line, and so forth. The information after the second column is written in the HTML language. If you know HTML, then you can[8] use that information.

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

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

`http://www.sagemath.org/doc/constructions/calculus.html?`

As you can see, we replaced `html/en/` with

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

`http://www.sagemath.org/doc/`

in order to construct that URL. That substitution will work in general. By the way, the `en` refers to the English language, with `fr` referring to French, `de` to German, and `ru` to Russian, according to the normal two-letter language codes in use on the internet.

1.11. Using Sage to Take Derivatives

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

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

to learn the answer

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

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

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

to receive the answer

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

Sometimes you might have earlier defined your own function by

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

earlier, so you can then type

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

to get the derivative, which is naturally

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

but you can also type

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

which is silent, but then you can type `gprime(2)` or `gprime(3)` and it does what you expect. It tells you the values of $g'(2)$ and $g'(3)$. You can also type

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

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

Sage can also do very difficult derivatives:

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

which provides the answer

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

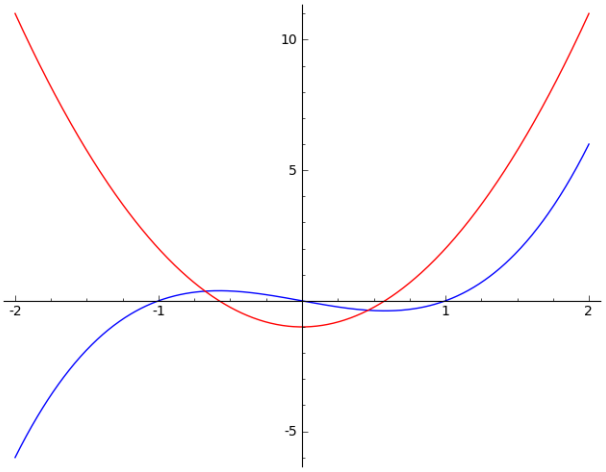
Now if you're interested in giving your calculus skills a workout, but possibly a very intense one, you can try to see how you would calculate the derivative of x^x with your pencil. By the way, it is important to note that `log` refers to the natural logarithm, not the common logarithm, as was explained on Page 5.

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

One of my favorite bits of code in Sage is

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

and I use this quite frequently. This plots $f(x)$ and $f'(x)$ on the same graph, with $f(x)$ in blue, and $f'(x)$ in red. You can very easily change what function you are analyzing by changing that first line. Here we are analyzing $f(x) = x^3 - x$, in blue, and its derivative, $f(x) = 3x^2 - 1$, in red. The plot is as follows:



1.11.2. Higher-Order Derivatives

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

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

to get the answer

```
6*x
```

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

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

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

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

which is more readable than writing `diff`.

1.12. Using Sage to Calculate Integrals

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

An Example of Integration:

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

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

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

On the other hand,

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

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

The Indefinite Integral:

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

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

you would simply type

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

and then you'd learn the answer is

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

Similarly, if you wanted to know

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

you would simply type

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

and then you'd learn the answer is

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

More About the +C:

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

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

when computing an indefinite integral.

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

With this in mind, you have to keep track of the $+C$s yourself. Correctly handling the $+C$s is a key task in one of the projects (the project about ballistic projectiles), in Section 2.5 on Page 79.

The Definite Integral:

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

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

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

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

and we are the answer

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

However, remember that "log" means the natural logarithm, not the common logarithm, as explained on Page 5.

Impossible Integrals:

The word "impossible" is a dangerous one in mathematics. Some integrals are famously impossible. The two most famous are the Fresnel Integral

$$\int_0^x \sin \frac{\pi t^2}{2} \, dt$$

which is important in optics, and the Gaussian Integral

$$\int_0^y \frac{2}{\sqrt{\pi}} e^{(-x^2)} \, dx$$

which is pivotally important in probability. What do we mean by "impossible" here? There is no function, built up of addition, subtraction, multiplication, division, roots, exponents, logarithms, trigonometric functions, and inverse trigonometric functions[9] which will have, as its derivative, either the Fresnel or the Gaussian.

Yet, in applied mathematics, we need to calculate these integrals! In particular, the Gaussian Integral is superbly important in statistics. So what is done, is that the definite integral can be computed by dividing the interval into many, many tiny regions. Lots of narrow rectangles and trapezoids can be used to approximate the curve. The numerical areas of those rectangles and trapezoids are added together, to give a numerical approximation for the integral. (That's why it is called "numerical integration.") Each of these rectangles or trapezoids might well have a small amount of error, but when you add up all the areas, the hope is that the errors cancel out a bit. You probably did this technique during your calculus class at some point, or will at some future point.

Even more sophisticated methods for numerical integration include using polynomials instead of trapezoids, and Sage uses extraordinarily intricate techniques to make the best approximation possible. These approximations are an old and deep subject, going back to Simpson's Rule, where Simpson used tiny parabolas instead of narrow trapezoids. This allowed him to get rather accurate approximations using far fewer computations than the trapezoidal rule would require. Since he was working in the first half of the 1700s, that's quite an accomplishment.

The above might have been a bit confusing, but the whole point of Sage is that it will worry about these details for you, so long as you know what you want from Sage.

Let's take a specific example. Consider the following function

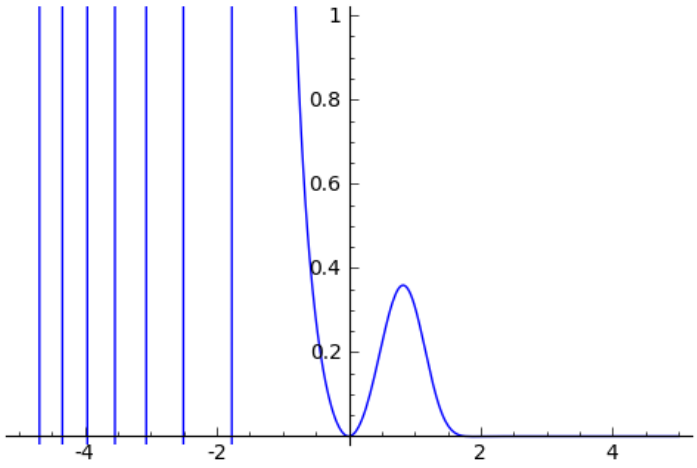
$$f(x) = e^{-x^3} \sin(x^2)$$

which is an easily understood and continuous function. We can graph it with the commands

```
plot( exp( -x^3 ) * sin(x^2), -5, 5, ymin = 0, ymax=1 )
```

We obtain the following plot

[9] We should also include the hyperbolic cousins of the trigonometric functions and inverse trigonometric functions, which you might or might not have been taught about.



This function $f(x)$ seems interesting, but computing its integral is quite frustrating. I cannot solve that integral with my pencil, and I am willing to wager that you cannot either. Try it if you like.

While it is the case that there is some function $g(x)$, somewhere, which has $f(x) = e^{-x^3}\sin(x^2)$ as its derivative—a logical consequence of some advanced real analysis and the fact that f is continuous—the practicalities of the matter are that there is no way to write down $g(x)$ as a formula. To be really precise, there is no function $g(x)$, built up of addition, subtraction, multiplication, division, roots, exponents, logarithms, trigonometric functions, and inverse trigonometric functions (as well as their hyperbolic cousins, which you might or might not have been taught about) which will have $g'(x) = f(x)$.

In other words $f(x) = e^{-x^3}\sin(x^2)$ is integrable, but the integral cannot be written in terms of common (and not so common) functions. However, one could produce a $g(x)$ made with an infinite series in Big-Sigma notation that would have $g'(x) = f(x)$.

Returning now to Sage, since

$$\int e^{-x^3}\sin(x^2)dx$$

does not have a nice expression, then when you type

```
integral( exp(-x^3)*sin(x^2),x)
```

the Sage replies not with an answer, but instead with

```
integrate(e^(-x^3)*sin(x^2), x)
```

which represents an admission of defeat. However, have no fear, because we can calculate numerical answers approximately—with very high accuracy—and we'll learn about that momentarily.

Numerical Integration in Sage:

A nefarious integral, famous among calculus teachers, is

$$\int t^{20} e^t dt$$

where the 20 could be any decently large number. This comes up in explaining what Euler's gamma function is, but we aren't too concerned with that right now. In any case, we could just type

```
integral( (t^20)*(e^t), t )
```

but then we get back

```
(t^20 - 20*t^19 + 380*t^18 - 6840*t^17 + 116280*t^16 - 1860480*t^15
+27907200*t^14 - 390700800*t^13 + 5079110400*t^12 - 60949324800*t^11
+ 670442572800*t^10 - 6704425728000*t^9 + 60339831552000*t^8
- 482718652416000*t^7 + 3379030566912000*t^6 - 20274183401472000*t^5
+ 101370917007360000*t^4 - 405483668029440000*t^3
+ 1216451004088320000*t^2 - 2432902008176640000*t
+ 2432902008176640000)*e^t
```

which is large enough that it isn't clear how to get an idea of what that means. If instead, we had certain bounds, such as to calculate

$$\int_2^3 t^{20} e^t dt$$

then we would type

```
integral( (t^20)*(e^t), t, 2, 3 )
```

and get instead

```
-329257482363600896*e^2 + 121127059051462881*e^3
```

which is getting toward an answer! If you really want a numerical answer then you should use our old trick, the command $N()$, and type

```
N( integral( (t^20)*(e^t), t, 2, 3 ) )
```

which would return the number

```
8.79797452800000e9
```

and that means $8.79797452800000 \times 10^9$. This highlights the difference and similarities of the various types of answer. If you've forgotten about $N()$, see Page 3.

That number which we just calculated looks like an integer, because it ends with 528 when you carry out the 10^9 part—but when we look at the exact answer, we can see that no, it is not an integer. We know that because e^2 and e^3 are involved, and e is an irrational number. Whenever dealing with computations, it is critical to remember what is an approximation, and what is exact. Irrational number can be expressed exactly on occasion (like we just did here, using e^2 and e^3) but that is the exception and not the rule. In any case, this paragraph has nothing to do with Sage, therefore we return to the topic at hand.

Speaking of approximations, you can also jump to the numerical integral very rapidly. I'm sure we can both calculate

$$\int_0^1 \left(x^3 - x\right)\, dx = \left(\frac{1}{4}x^4 - \frac{1}{2}x^2\right)\Big|_0^1 = \left(\frac{1}{4}1^4 - \frac{1}{2}1^2\right) - \left(\frac{1}{4}0^4 - \frac{1}{2}0^2\right) = \frac{1}{4} - \frac{1}{2} = -\frac{1}{4}$$

However, if you wanted to ask Sage to calculate that integral numerically, then you must type

```
numerical_integral( x^3 - x, 0, 1)
```

which will output

```
(-0.24999999999999997, 2.775557561562891e-15)
```

where the first number is the best guess Sage has for the answer, while the second number is the uncertainty. In this case, the uncertainty is 2.77 quadrillionths—which is very impressive.

Of course, Sage can do this integral exactly as well, using the commands we learned moments ago. What is interesting about numerical integration are the cases when you cannot do the indefinite integral, because like the Fresnel Integral above it cannot be written, but you can find good numerical estimates. Consider again

$$\int_1^3 e^{-x^3} \sin(x^2)\, dx$$

but now with bounds, so that it can be done numerically. We would type

```
numerical_integral( exp(-x^3)*sin(x^2),1,3)
```

and that returns back

```
(0.077997011262087815, 8.684049685199503 7e-16)
```

with (as before) the first number being the answer, and the second the uncertainty. In this case, the uncertainty is 868 quintillionths—which is very impressive. Another way to say it is that the uncertainty is (1/1152) trillionths.

Integration by Partial Fractions:

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

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

then you must do

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

followed by

```
f.partial_fraction()
```

and get the answer

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

which is correct. In fact, we just calculated

$$\frac{x^3 - x}{x^2 + 5x + 6} = x - \frac{6}{x+2} + \frac{24}{x+3} - 5$$

However, you can also do the shortcut and ask instead

```
integral( (x^3-x)/(x^2+5*x+6), x )
```

which will result in

```
1/2*x^2 - 5*x - 6*log(x + 2) + 24*log(x + 3)
```

Improper Integrals:

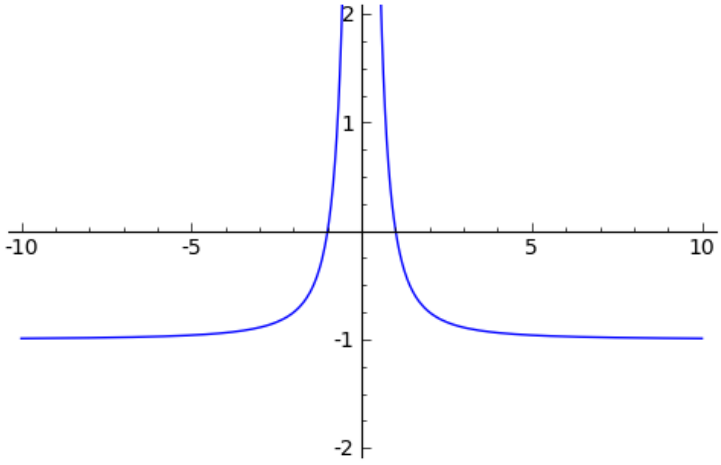
While the following integral is mathematically valid

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

and the right-hand side can be evaluated at $x = -1$ and $x = 1$, there is no finite answer to

$$\int_{-1}^{1} \left(-1 + \frac{1}{x^2}\right)\, dx$$

because this improper integral is divergent. The graph will help reveal why that is the case. Here is the plot of $-1 + 1/x^2$



generated by

```
plot( -1+1/x^2, -10, 10, ymin=-2, ymax=2 )
```

Yet, Sage knows this and will say

```
-x - 1/x
```

in response to

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

but instead will respond

```
ValueError: Integral is divergent.
```

to the inquiry

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

If you'd like a proof of the fact that the above integral is divergent, consider that

$$\int_{-1}^{-1} \left(-1+\frac{1}{x^2}\right)\,dx = \int_{-1}^{0} \left(-1+\frac{1}{x^2}\right)\,dx + \int_{0}^{1} \left(-1+\frac{1}{x^2}\right)\,dx$$

Let's consider the integral above, from 0 to 1, by means of computing that integral from a to 1, where a will be slightly more than 0.

$$\begin{aligned}\int_a^1 \left(-1+\frac{1}{x^2}\right) &= \left(-x-\frac{1}{x}\right)\Big|_a^1 \\ &= \left(-1-\frac{1}{1}\right)-\left(-a-\frac{1}{a}\right) \\ &= a-2+\frac{1}{a}\end{aligned}$$

As you can see, when a approaches 0 from the positive real numbers, the value of $a-2+1/a$ becomes huge. More precisely, we can write

$$\lim_{a\to 0^+}\left(a-2+\frac{1}{a}\right)=\infty$$

but then that also means

$$\lim_{a\to 0^+}\int_a^1 \left(-1+\frac{1}{x^2}\right)\,dx=\infty$$

and, as it turns out, the same is true for the integral from -1 to 0 as well. Therefore, the original integral is equal to $\infty+\infty=\infty$.

More Improper Integrals:

Another type of improper integral is

$$\int_2^{\infty} \frac{1}{x^2}\,dx$$

and the way to write that in Sage is to type

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

to which Sage responds 1/2, or another example is

$$\int_{-\infty}^{\infty} e^{-x^2}dx$$

which in Sage is

```
integral(exp(-x^2),x,-oo,oo)
```

and it gives the correct response of $\sqrt{\pi}$.

The idea is simply that `oo` is a special code for "infinity." It is two letter o's, and the idea is that if you squint, `oo` looks like an infinity symbol.

The Function erf and Integrals:

Among mathematicians and especially among statisticians, the integral

$$\int_0^y \frac{2}{\sqrt{\pi}} e^{(-x^2)}\, dx = \text{erf}(y)$$

is extraordinarily important. This is the Gaussian Integral that I told you about earlier; in particular, I said that this integral cannot be found using the elementary functions of mathematics. It is so important that it has a name, and that name is `erf`. While being friendly and pronounceable, `erf` stands for the "Error Function."

Thus if you type into Sage

```
integral( (2/sqrt(pi)) * exp(-x^2), x, -oo, 2)
```

it will respond

```
(sqrt(pi)*erf(2) + sqrt(pi))/sqrt(pi)
```

which is correct because

$$\begin{aligned}\int_{-\infty}^2 \frac{2}{\sqrt{\pi}} e^{(-x^2)}\, dx &= \int_{-\infty}^0 \frac{2}{\sqrt{\pi}} e^{(-x^2)}\, dx + \int_0^2 \frac{2}{\sqrt{\pi}} e^{(-x^2)}\, dx \\ &= 1 + \text{erf}(2) = \frac{\sqrt{\pi}}{\sqrt{\pi}}(1 + \text{erf}(2)) = \frac{\sqrt{\pi}\text{erf}(2) + \sqrt{\pi}}{\sqrt{\pi}}\end{aligned}$$

Perhaps you might find that $1 + \text{erf}(2)$ is a more compact and simpler answer. I would be inclined to agree. I'm not quite sure why Sage does not simply its final answer further.

In any case, the point is that there are some functions out there which do not have integrals writable using elementary functions, but, which Sage calculates anyway, using the `erf` function.

The Assume Command, and Integrals

This example was identified by Joseph Loeffler, who brought it to my attention. Let's say that you wanted to evaluate the integral

$$\int_1^x \frac{t^3 + 30}{t}\, dt$$

and that you typed, accordingly, the Sage commands:

```
var("t")
integrate( (t^3+30)/t, t, 1, x)
```

(By the way, `integrate` is a synonym for `integral`.) As it turns out, you'd receive back a response that is a huge error message. We must remember that in Sage, the last lines of an error message are the ones that are important. The last lines are as follows:

```
ValueError: Computation failed since Maxima requested additional
constraints; using the 'assume' command before integral
evaluation *may* help (example of legal syntax is 'assume(x-1>0)',
see `assume?` for more details)
```

```
Is x-1 positive, negative or zero?
```

The complaint here, sent to Sage from Maxima, is justified because the function $f(t) = (t^3 + 30)/t$ will explode at $t = 0$, because it attempts to divide by zero. The integral is therefore improper if 0 is found between 1 and x. However, if $x - 1 > 0$, or in plain English, if $x > 1$, then that's clearly not going to happen. Therefore, Sage is very justified in asking us to explicitly state this assumption.

Let's follow Sage's recommended course of action by adding the assume command before our `integrate` command. We now have

```
var("t")
assume( x>1 )
integrate( (t^3+30)/t, t, 1, x)
```

which produces the response

```
1/3*x^3 + 30*log(x) - 1/3
```

as expected.

The `assume` command is rare enough that many seasoned Sage users do not know that it exists. It is unlikely to come up in your work. However, we'll see another example of it on Page 184, in a summation, and on Page 202 in an integral related to a Laplace Transform.

1.13. Sharing the Results of Your Work

In this section, I'll share with you seven effective ways of sharing the results of your work in Sage. These can be great for scientific collaboration, submitting work to a faculty member, getting help from a friend, or just explaining a math concept to someone. My two most favorite methods are accessed via the "Share" button on the Sage Cell Server screen.

When you have work in the Sage Cell Server, you can click on the "Share" button. Then you will see options for making a short temporary link, a permalink, and a "2D bar code," also called a "QR code." The permalink is my favorite. It will generate an enormous URL, and prove to you that the new URL works by reloading the page with that URL. Then you should highlight that URL from the web-browser, from the spot where you would normally type `https://sagecell.sagemath.org/`, taking great care to make sure that you've highlighted the whole thing. This URL can now be used in documents, in emails, in chats, or even on Facebook. The URL actually encodes the Sage code in the window, and thus your code is not stored anywhere and therefore cannot be lost, deleted, or destroyed. As long as you retain a complete and unmodified URL, then that URL will always work, even years later.

The permalink is long and ugly. This should not be surprising, because it actually encodes your entire Sage block of code. In fact, the permalink is so long that it can cause problems in some browsers or email processors. In almost all cases, it is longer than 140 characters and thus cannot be sent in

a text message or be tweeted. The short temporary link, on the other hand, is shorter and less intimidating to a human. Being shorter, it is less likely to be truncated in transit when sent by various means. However, it does not contain a complete copy of your code. Instead, your code is stored on the server, and the URL retrieves it. There is no guarantee for how long the short temporary link will work. Perhaps a day, perhaps a year, or perhaps forever but perhaps only an hour. In my experience, they often work several days after being generated. Overall, I'd have to say that if the length and ugliness are not a problem, then the permalink should always be used and the short link should never be used.

Those are my two favorite methods of sharing Sage code. The third method I'd like to share with you is the large 2D barcode produced when you click "share." This actually encodes the permalink and thus will work with all smartphones that can read those 2D barcodes. Most phones will require you to install an app for the smartphone to be able to handle those barcodes, which are sometimes called "QR codes," "Quick Response Codes," or "matrix barcodes." On my phone, the app[10] was called "QR Code Reader," and was free. If you photograph the barcode using one of those apps and the camera inside your smartphone, then it should open up a web browser and present the same results as the permalink will. You don't even need to click "Evaluate"! Here is a sample 2D barcode:

The fourth method is when you've made a plot (as we learned in Section 1.4 on Page 8). If you'd like to include this image in any sort of document, or email it to someone, then just save the image file from your web browser—in the same way that you'd save any other image. In most browsers, this is a right-click followed by selecting "Save Image As...," and entering in a filename at the appropriate spot. This image will be in the

[10]You really do need the app, however. Otherwise you just get a photograph of the QR Code, which is totally unexciting.

"portable network graphic" or `*.png` format. It will work in nearly all programs that allow the importing of an image. Alternatively, some high-end mathematics and physics journals prefer "encapsulated postscript" or `*.eps` files. Those can be generated by

```
P = plot( x^3-x, -2, 2 )
P.save("myplot.eps")
```

which will give you a link that will start the downloading of `myplot.eps` when clicked upon.

The fifth method is just highlighting the Sage code, copying, and then pasting it into an email. Sometimes I will paste it into SageMathCloud, or my university's course management software, D2L. Emailing code can save time, and it is easy to post code in this way.

The sixth method is to print a screenshot of the Sage Cell Server. Using a bit of scrolling around and resizing, one can often cause the entire Sage block of code and its output to be visible at the same time. Then you can use your browser's print command (often under the file menu) to print the webpage just like you'd print any other webpage. (Often the web browser's print command is under the "File" menu.) This is often how I ask students in my classes to submit a homework problem that has been done in Sage. Last but not least, the seventh method is to follow the steps for printing a screenshot of the Sage Cell Server, and instead of printing it as a physical paper document, to select "Save as a PDF" from the web browser's print menu. This PDF is essentially identical as a document to what you'd get from printing, except that it is an electronic copy, rather than a paper copy.

Scientific collaboration is one of the great joys of working in a STEM[11] subject. I hope that you will have frequent opportunities to share not only your ideas, but the hardcore mathematical and quantitative details of your work, with others. Since Sage has been developed by hundreds of people scattered across the globe, it is unsurprising that there many features built-in to Sage to enable sharing. SageMathCloud has even more features, including organized projects, sharing on an invitation-only basis, and live-chat features. You can learn about those on the SageMathCloud website.

[11]STEM: Science, Technology, Engineering and Mathematics.

Chapter 2

Fun Projects using Sage

Here are some extended examples of using Sage to explore a non-trivial problem in an area outside of computer algebra. We present one example from each of several disciplines. The idea is that these projects would each take a typical student about one long weekend to complete.

Some of these projects can be explored using information taught in Chapter One, and do not require any additional familiarity with Sage. Some projects do require a bit more, but that is clearly marked in the first few paragraphs of the project description.

Microeconomics: Modeling the effect of the selling price on the sales of ice cream, in terms of a demand function, a revenue function, a cost function, and a profit function

Biology: Modeling clogged arteries and Poiseuille's Law, and the catastrophic effects of even small percentages of blockage on blood flow. No knowledge of biology is assumed

Industrial Optimization: Solving a linear system of inequalities to optimally route Taconite from 7 mines to 4 ports for shipping, while minimizing shipping costs and respecting capacities

Chemistry: A method for balancing rather complicated chemical reactions by using the RREF of matrices

Physics: Solving ballistic projectile motion problems in Sage, including counter-battery fire

Cryptology: Using Pollard's $p-1$ Method of factoring integers, to try to break the RSA cryptosystem

Pure Mathematics: Using differential calculus to analyze a quintic polynomial's features, for finding the optimal graphing window

Electrical Engineering: A mini-project on generating electric-field vector plots for the field generated by four charged particles in the coordinate plane.

Also, I am writing a book for a course commonly called *Finite Mathematics* in the USA, and *Quantitative Methods* in the UK. This is a course for people majoring in business, accounting, finance, economics, marketing,

management, and so forth. A classic topic in that course is the Leontief Input-Output Analysis method of modeling how various sectors of the economy interact. This would be a topic in macroeconomics, to complement the microeconomics project given here. While it does require a bit of matrix algebra, Leontief Analysis is very well suited to Sage and relatively easy to pick up. You can find a version of that project on my webpage: `http://www.gregorybard.com/`

2.1. Microeconomics: Computing a Selling Price

I was once told that the goal of the first microeconomics course should be twofold: First, to teach the concept of "supply and demand." Second, to teach that the three actions of "maximizing profit," versus "maximizing revenue," or "minimizing costs," are three extremely different goals. This is an oversimplification, of course, and was probably meant as a joke. Nonetheless, it highlights the paramount importance of these two topics.

This project will look closely at the issue of "maximizing profit," versus "maximizing revenue," or "minimizing costs." It is an extended study of the micro-economics of the sale of ice cream. We imagine a vendor with an ice-cream cart at a busy intersection in Central Park in New York City.

The ice cream vendor observes that when he charges \$ 3 per ice cream cone, he only sells 120 cones per day. On the other hand, when he charges \$ 2 per ice cream cone, he sells 200 cones per day. In our example, because he has long since paid off his start-up expenses, the costs of doing business are just 50 cents per cone, plus a \$ 100 per day fee for his permit to sell food in Central Park. As you might imagine, if he makes his price too high, he will sell too few cones, and won't make much profit; if he makes his price too low, he will sell more cones, but he still won't make much profit.

The ice cream vendor is going to assume that the demand varies linearly with price. This often is not quite true, but it is often a reasonable approximation. There are several different ways to realize that the demand n, and the price p, will have the relationship

$$n = 360 - 80p$$

or equivalently

$$p = 4.50 - \frac{n}{80}$$

Let's pause a moment, and see if this makes sense. First, plugging in $n = 200$ and $n = 120$, we do get the prices of \$ 2 and \$ 3 that we expect. If we set $p = 2.50$, then we get 160, as we would anticipate—that's the midpoint.

The easy way to find those equations is to compute the slope connecting the points $(200, 2)$ and $(120, 3)$, and then find the equation of the line between those two points. Of course, the form of the equation depends on

whether or not you assign p or n the role of x. I have seen both conventions used in textbooks—there is no "industry standard."

Now observe that, at a price of \$ 4.50, he will sell no ice cream at all. This should be believable, if people are usually expecting to pay around \$ 2, then you cannot charge them more than double what they are accustomed to pay. The price where demand is zero is called "the maximum feasible price." Meanwhile, if the price of a cone is \$ 0, then we sell 360 cones per day—that's called the saturation point and it is the most that one could ever hope to sell, namely the demand at the price of \$ 0. The model breaks down for prices over \$ 4.50, because it predicts a sale of a negative number of cones, which does not make sense. Likewise, if you set a demand above 360 cones per day (the "saturation point") then a negative price is called for, which also does not make sense. So the model is valid for $0 \leq p \leq 4.50$ dollars, or equivalently $0 \leq n \leq 360$ cones.

We'd now like to make some revenue, cost, and profit functions in terms of the number of cones sold, n. Clearly if one sells n cones at price p, then the revenue is np. Therefore, we have

$$r(n) = np = n\left(4.50 - \frac{n}{80}\right) = 4.50n - \frac{n^2}{80}$$

as the revenue function. Meanwhile, the cost function is 50 cents per cone, plus \$ 100, for a total of

$$c(n) = 0.50n + 100$$

and that gives us a profit function of

$$p(n) = \frac{-n^2}{80} + 4n - 100$$

which is a parabola. It makes sense that we should get some sort of curve, because if you make the price very low, or very high, then the profit is zero or worse—you have a loss. Somewhere in the middle is the sweet spot, where profit is made.

Let's make a table and see that these functions make sense and match our prior data:

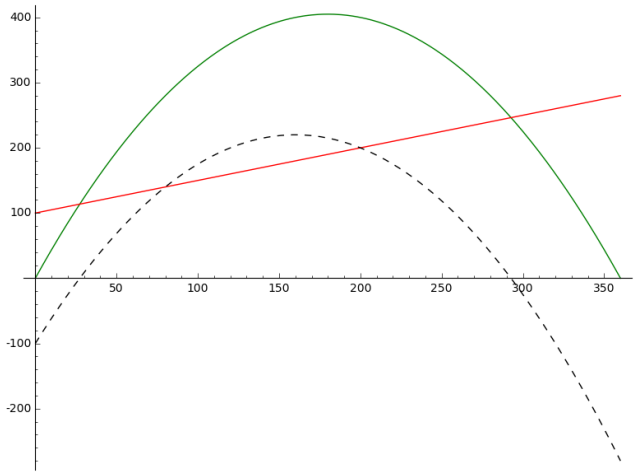


FIGURE 1. The Plot for the Microeconomics Project

Price	Cones Sold	Revenue	Costs	Profit
$ 0.00	360	$ 0.00	$ 280.00	$ -280.00
$ 0.50	320	$ 160.00	$ 260.00	$ -100.00
$ 1.00	280	$ 280.00	$ 240.00	$ 40.00
$ 1.50	240	$ 360.00	$ 220.00	$ 140.00
$ 2.00	200	$ 400.00	$ 200.00	$ 200.00
$ 2.50	160	$ 400.00	$ 180.00	$ 220.00
$ 3.00	120	$ 360.00	$ 160.00	$ 200.00
$ 3.50	80	$ 280.00	$ 140.00	$ 140.00
$ 4.00	40	$ 160.00	$ 120.00	$ 40.00
$ 4.50	0	$ 0.00	$ 100.00	$ -100.00

I recommend that you pick 2 or 3 values, and manually verify those 2 or 3 rows of the table, before continuing onward.

Okay, did you really verify 2 or 3 rows of the above? It is really useful to do so before reading further.

In any case, the graph of these three functions is given in Figure 1. As you can see, the profit function is the dashed parabola (in black if you're reading the color version of this book), the cost function is the line (in red if you're reading the color version of this book), and the revenue function is the solid parabola (in green if you're reading the color version of this book). There are lots of facts that can be gleaned from this:

- When the cost function and the revenue function cross, that's the break-even point. In other words, when the cost and revenue are equal, you break even. Here we have two break even points, one just under 30, and one just under 300. To find them, we type

```
find_root(  (-x^2)/80 + 4.5*x == 0.5*x+100, 250, 350)
```

and also

```
find_root(  (-x^2)/80 + 4.5*x == 0.5*x+100, 10, 50)
```

to find out that at roughly 292.664 cones and 27.3350 cones, we break even. Between these values is a profit, and outside these values, there is a loss. That seems to match the data in the table.

- The other definition of the break-even point is when profit is zero. So we could have tried to find the roots of the profit function, as an alternative means of finding the break-even points. We would do this by typing

```
find_root(  (-x^2)/80 + 4*x - 100, 10, 50)
```

and also

```
find_root(  (-x^2)/80 + 4*x - 100, 250, 350)
```

to find those same values (292.664 and 27.3350), and you'll see that the dotted/black curve crosses the x-axis there.

- The cost function is never zero. That's because the ice cream vendor has to pay the $ 100 per day fee. As you can see, the line never crosses the x-axis for any positive n.
- The revenue function is zero at two points. First, when no ice cream is sold—that makes sense. Surely we can agree that if no ice cream is sold, then there is no revenue. We found earlier that is at $ 4.50 per cone, and 0 cones per day. Second, revenue is $ 0 when the ice cream is sold for 0 cents per cone. Clearly, if you don't charge anything, then there is no revenue. That occurs at the saturation point, of 360 cones per day and $ 0 per cone. To find these in Sage, we should type

```
find_root( 4.50*x-(x^2)/80, 0, 50)
```

and also

```
find_root( 4.50*x-(x^2)/80, 300, 400)
```

to get the values 0.0 and 360.0.

- Now we've found when profit is zero and when revenue is zero, and determined that cost is never zero. The next step is to try to minimize or maximize these functions. If you've had calculus, then you know how to minimize or maximize a polynomial. If not, then you might recall the fact that a parabola $y = ax^2 + bx + c$ has its optimum at the point $x = -b/2a$.
- For the revenue function, we see that maximum revenue occurs at 180 cones, a cost of $ 2.25 per cone, and a revenue of $ 405. The costs would be $ 190, for a profit of $ 215.
- For the profit function, we see that maximum profit occurs at 160 cones, a cost of $ 2.50 per cone, and a revenue of $ 400. The costs would be $ 180, for a profit of $ 220.

- The point of minimum costs would be 0 cones of ice cream, for a cost of $ 100. However, the revenue is $ 0 and the profit is negative, a loss of $ 100. Definitely, this is not a good business plan. (Even if the costs are lower in this bullet than in the previous two bullets.)
- Did you notice how the point of maximum profit had lower revenue ($ 400 vs $ 405) but higher profit ($ 220 vs $ 215) than the point of maximum revenue? Did you notice how the point of maximum revenue had lower profit ($ 215 vs $ 220) but higher revenue ($ 405 vs $ 400) than the point of maximum profit?

In conclusion, we see that "maximizing revenue," "minimizing costs," and "maximizing profit," are three different objectives. Furthermore, we should suggest that the ice cream vendor sell his ice cream at $ 2.50 per cone—because maximizing profit is the only intelligent thing to do.

By the way, the code to produce the image with the graphs of the three functions is as follows:

```
r(x) = 4.50*x - x^2/80
c(x) = 0.5*x + 100
p(x) = r(x) - c(x)

P1 = plot( r(x), 0, 360, color='green' )
P2 = plot( c(x), 0, 360, color='red' )
P3 = plot( p(x), 0, 360, color='black', linestyle='--' )
P = P1 + P2 + P3

P.show()
```

Challenge

Your challenge is to analyze a new business situation. A particular book is pre-sold by a publisher in two test markets for $ 49.95 and for $ 39.95. It sells 80 copies and 140 copies respectively. In the past, the general sales for the year are 100 times the sales of one of the test markets. (Just to be clear, that means if the price is set at $ 49.95, the marketing experts predict 8000 copies will be sold, and likewise 14,000 copies at $ 39.95.)

The books require $ 7 to print, plus $ 4000 payable once for the proof-reading costs. The publisher gets 50% of the sales revenue. In case you are curious, usually around 20% of the sales revenue goes to the local book-store, 20% to the distributor, and 10% goes to the author's royalties. You may assume that the sales price and the number of copies sold have a linear relationship.

The deliverables for this project are

(1) an equation relating the sales price and the number of copies sold,
(2) the revenue function,
(3) the cost function,
(4) the profit function,

(5) the image showing all three functions graphed simultaneously,
(6) the break even point(s),
(7) the n and p to result in maximum revenue,
(8) the n and p to result in maximum profit.

2.2. Biology: Clogged Arteries and Poiseuille's Law

If an American dies of natural[1] causes, the probability it is of heart disease is 35.30%, of various forms of cancer is 33.94%, and of any other cause is 30.75%. Considering that heart disease kills roughly a third of us, the realities of clogged arteries and cholesterol should concern us all.

Blood flowing through an artery is governed by the same equation as water flowing in a pipe or oil flowing through a hose. This applies to all such flows of fluids through a cylinder, except for turbulent flow (where the fluid is churning around and tumbling over itself) and compressible flow (such as air flowing in a pneumatic hose). The equation is called Poiseuille's Law, named for Jean-Lonard Marie Poiseuille (1797–1869), whose dissertation[2] was applying this mathematics to the human aorta.

In this project, we're going to explore the relationships between some of the variables in Poiseuille's Law. We are going to display those interdependencies through tables. In order to do this project, you need to have read Section 5.1, which describes how to make tables in Sage.

The Background

The formula itself is as follows.

$$\dot{V} = (\Delta P)\frac{\pi r^4}{8\mu L}$$

and regrettably, we do not have time to derive it here. However, I strongly recommend the curious reader to refer to the article "Blood Vessel Branching: Beyond the Standard Calculus Problem," by Prof. John Adam, published in the *Mathematics Magazine*, Vol. 84, by the Mathematical Association of America, in 2011.

Let's take a moment to identify the meaning of each of those variables.

- The $\dot{V}$ is the volume flow rate, in units such as cubic centimeters per minute. In a major artery, a typical flow might be 100 cm^3/minute.
- The radius of the artery is r, in units such as centimeters. A major artery might be 0.3 cm or very roughly 1/8th of an inch in diameter.
- The length of the artery is L, again in units such as centimeters. A major artery might be 20 cm or 8 inches in length.

[1]Based on 2010 data, from the Center for Disease Control and Prevention's website "FastStats." This data excludes (as unnatural causes) any homicides, suicides, and accidents.

[2]Poiseuille, J-L. M., *Recherches sur la force du coeur aortique*, D.Sc., *École Polytechnique*, Paris, France. 1828.

Note: The unit of pressure can vary. The standard unit in the American system of units is pounds per square inch, because forces are measured in pounds. Likewise, in the metric system, because forces are measured in Newtons, the pressure units then become Newtons per square meter. Using Newtons per square meter is kind of funny because arteries aren't several square meters in area. Another unit of pressure used is "atmospheres" where atmospheric pressure is 1 atmosphere. That's 14.6959 pounds per square inch or 101,325 N/m^2.

- Medicine has been around for a long time. For historical reasons, it is normal to measure blood pressure in a very archaic unit, called "millimeters of Mercury." When you go to the doctor and she tells you that your blood pressure is "120 over 80," that 120 means "120 millimeters of Mercury," abbreviated 120 mmHg. Just take it as a unit, and consider 120 to be "normal." If you have 145 mmHg for the first number (called systolic), you might be diagnosed as having "hypertension." A score of 180 mmHg systolic can result in organ damage and is called a "hypertensive emergency."
- Viscosity is a physics variable that you might or might not have seen before, and is denoted μ. It represents how "sticky" or "thick" a fluid might be. Since blood is the fluid in question, and we won't consider other fluids, we can just take μ to be a constant. The usual unit is a "poise." A typical value for blood might be $\mu = 0.0035$ poise. In comparison, the oil in your car might have $\mu = 0.250$ poise (because it is sticky) and water has $\mu = 0.001$ millipoise (because it is not sticky). Honey is very sticky, and has a viscosity of 20 poise. We can't use the poise as our unit, because we're using an antique unit for pressure. For us,

$$\mu = 4.37535\cdots \times 10^{-8}$$

is just a constant in the formula. The units of viscosity turn out to be "sec mmHg," which is extremely strange, but that is what will make the correct units work out in the formula.

Your Challenge

For this project, you're going to create some tables that can be used to help a physician understand the numerical realities of that quartic relationship in Poiseuille's Law between arterial radius and blood flow rate.

The tables will allow some variables to change, while most variables are locked at "standard values" that are chosen to be very typical. The standard values for our variables in this project are $\mu = 4.375359 \times 10^{-8}$ for the viscosity, $L = 20$ cm for the artery length, $r = 0.3$ cm for the arterial radius, $\dot{V} = 100$ cubic centimeters per second for the blood flow rate, and finally ΔP is 0.0275 mmHg.

- Your first table should keep our standard values for μ, L, ΔP, and compute $\dot{V}$ based on r, using Poiseuille's Law. The r should be computed based upon the percent blockage, from 0% up to 50%, in 2% increments. The first column of the table should be the percent blockage, the second column should be the blood flow rate or $\dot{V}$.
- Your second table should keep our standard values for μ, L, and $\dot{V}$. Again, the r should be computed based upon the percent blockage, from 0% up to 50%, in 2% increments. This time, you will compute the ΔP required to achieve the flow rate of 100 cc/minute given the blockage (the $\dot{V}$ given the ΔP), using Poiseuille's Law. The first column of the table should be the percent blockage, the second column should be the ΔP. The third column should represent ΔP as the "percent of normal." In other words, 3/2 the normal ΔP should be reported as 150%.
- The third table should keep our standard values for μ, L, and ΔP. The first column should be the "percent normal flow" from 100% down to 25%, in steps of 3%. The second column should be the flow rate, in cubic centimeters per second, identified by that percentage. The third column should be the radius implied by this, as computed by Poiseuille's Law. The fourth column should be the percent blockage. (In other words, 0.15 cm radius is a 50% blockage because $0.15/0.3 = 0.5$.)

You can check your work by plugging into the original equation, but you might also find it useful to compare your first table to the following, which is given in increments of 6%.

```
0 % blockage implies  99.9617636500122 cc/minute.
6 % blockage implies  78.0450430095128 cc/minute.
12 % blockage implies  59.9466058383290 cc/minute.
18 % blockage implies  45.1948885141475 cc/minute.
24 % blockage implies  33.3494195216211 cc/minute.
30 % blockage implies  24.0008194523679 cc/minute.
36 % blockage implies  16.7708010049720 cc/minute.
42 % blockage implies  11.3121689849831 cc/minute.
48 % blockage implies  7.30882030491648 cc/minute.
54 % blockage implies  4.47574398425329 cc/minute.
```

2.3. Industrial Optimization: Shipping Taconite

In this project, you will solve a very simple but realistic shipping-route problem, using Linear Programming. This project assumes that you have read Section 4.21 on Page 187, about how to do linear programming in Sage.

In particular, a mining conglomerate has seven mines around Minnesota and Wisconsin, and they have shipping facilities at Two Harbors, Green Bay, Minneapolis, plus a shipping complex at the twin ports of Duluth, MN

Shipping Costs

	Duluth, MN Superior, WI	Two Harbors, Minnesota	Green Bay, Wisconsin	Minneapolis, Minnesota	Total Production
Mine 1	$ 18/ton	$ 30/ton	$ 74/ton	$ 35/ton	8,000 tons/wk
Mine 2	$ 73/ton	$ 21/ton	$ 50/ton	$ 65/ton	13,000 tons/wk
Mine 3	$ 37/ton	$ 93/ton	$ 67/ton	$ 61/ton	7,000 tons/wk
Mine 4	$ 24/ton	$ 95/ton	$ 63/ton	$ 35/ton	16,000 tons/wk
Mine 5	$ 38/ton	$ 77/ton	$ 80/ton	$ 71/ton	13,000 tons/wk
Mine 6	$ 73/ton	$ 15/ton	$ 39/ton	$ 68/ton	8,000 tons/wk
Mine 7	$ 48/ton	$ 51/ton	$ 69/ton	$ 14/ton	9,000 tons/wk

TABLE 1. The Table of Shipping Costs and Mining Output

and Superior, WI. You will figure out how much taconite to ship from each mine to each of the four shipping facilities. You have several constraints.

First, it must be the case that the amount of taconite leaving each mine (for all four destinations) should be exactly the amount produced that week. It should not be greater—otherwise you're ordering your mines to ship more taconite than they actually have; it should not be less—otherwise you'll have a build up of taconite that cannot be stored on site. The total production at each mine is listed in Table 1.

Second, the demand and capacity of each shipping facility must be respected. The Two Harbors, WI facility can receive and ship up to 18,000 tons. The Green Bay facility can receive and ship up to 25,000 tons. The Duluth/Superior shipping complex can receive and ship between 15,000 and 45,000 tons, the minimum being there to ensure there is enough revenue to pay the bills at this expensive location. The Minneapolis facility can receive and ship between 10,000 and 30,000 tons.

Third, the shipping schedule should carry out the task of bringing the taconite to the receiving and shipping stations at minimum cost. The cost of shipping from each mine to each receiving and shipping facility is given in the Table 1, in dollars per ton.

Your goal is to model this problem as a linear program, code it up in Sage, and solve it. To help you out, I offer the following hint. Let $d_1, d_2, \ldots, d_7$ be the amount shipped out of Duluth/Superior from the seven mines. Let $t_1, t_2, \ldots, t_7$ be the amount shipped out of Two Harbors. Let $g_1, g_2, \ldots, g_7$ be the amount shipped out of Green Bay. Let $m_1, m_2, \ldots, m_7$ be the amount shipped out of Minneapolis.

Because you'll be showing a lot of work in this process, I have no fear of giving you the final answer. Here is the output of my program:

```
The minimum cost shipping arrangement is
1883000.0

The quantities shipped to Duluth/Superior from the seven mines are
{1: 8000.0, 2:0.0, 3: 7000.0, 4: 15000.0, 5: 13000.0, 6:0.0, 7:0.0}
```

```
The quantities shipped to Two Harbors from the seven mines are
{1: 0.0, 2: 13000.0, 3: 0.0, 4: 0.0, 5: 0.0, 6: 5000.0, 7: 0.0}

The quantities shipped to Green Bay from the seven mines are
{1: 0.0, 2: 0.0, 3: 0.0, 4: 0.0, 5: 0.0, 6: 3000.0, 7: 0.0}

The quantities shipped to Minneapolis from the seven mines are
{1: 0.0, 2: 0.0, 3: 0.0, 4: 1000.0, 5: 0.0, 6: 0.0, 7: 9000.0}
```

By the way, if you think this problem is complicated, please keep the following in mind. A realistic problem might contain many intermediate collection points, have several dozen mines, and have limited capacities on all of the roads between these points. It is easy to imagine a problem with 30 mines, 12 intermediate collection sites, and 5 ports, for $360 + 60 = 420$ routes. Each route would have a cost and a capacity limit. Each capacity limit is, itself, an inequality. Therefore, these sorts of problems, "in the real world," can get to be enormous.

2.4. Chemistry: Balancing Reactions with Matrices

When you take a class in chemistry, there's lots of fun things to do and learn. One of those things is balancing a chemical equation, which is often easy—however, a few chemical equations are rather hard (for non-chemists) to balance. Here, we'll learn to quickly and easily balance even the most hideous examples by using matrix algebra.

All the examples throughout this project, as well as the original idea of the project itself, are entirely due to Prof. Gergely Sirokman, of the Wentworth Institute of Technology in Boston, Massachusetts. I thank him for the time he spent crafting these examples for this book.

This project requires knowledge of how to handle linear systems of equations with infinitely many solutions. Appendix D provides complete coverage of that topic, but some readers will already know about that and will not need to read Appendix D.

2.4.1. Background

Let's start with an energetic example. Back in the age of airships, American airships were often filled with helium. However, Germany had no access to large supplies of helium, and so they used hydrogen instead. This is unfortunate because hydrogen gas can explode in the presence of a spark. The Hindenburg was an airship made by the Zeppelin Company, and exploded into a fireball in 1937 near Lakehurst, New Jersey. The hydrogen gas combines with oxygen in the air and forms water vapor. Here is the chemical formula for that reaction:

$$2\,H_2 + O_2 \to 2\,H_2O$$

Let's review what it means for a chemical equation to be balanced. If we have two molecules of H_2 and one molecule of O_2 before the reaction, then we have 4 hydrogen atoms and 2 oxygen atoms before the reaction. Then, if we have two molecules of H_2O after the reaction, then we have 4 hydrogen atoms and 2 oxygen atoms after the reaction. No atoms of hydrogen were created or destroyed during the chemical reaction, because we had 4 before and 4 after; likewise, no atoms of oxygen were created or destroyed because we have 2 before and 2 after. Since atoms are never created or destroyed in chemical reactions, we would have to reject this chemical equation as unbalanced if we did not have the before-number and the after-number matching. The match must be there for each and every atom in the chemical reaction—if even one before-after pair is unequal, then that means there is a mistake.

The burning of acetylene in a blow torch is a chemical reaction in which acetylene (C_2H_2) and oxygen from the air (O_2) combine to form carbon dioxide (CO_2) and water (H_2O). Based on those chemical formulas, we have to find the positive integers for the following 4 question marks below in order to balance the chemical equation.

$$? \ C_2H_2 + ? \ O_2 \rightarrow ? \ CO_2 + ? \ H_2O$$

Take a moment to try it now yourself, with a little bit of fiddling, and some old-fashioned guess and check. Try to find values for the ? marks.

No really, I mean it, give it a try.

Okay, by this point, I hope you have made an earnest attempt and have found the values to balance the equation:

$$2 \ C_2H_2 + 5 \ O_2 \rightarrow 4 \ CO_2 + 2 \ H_2O$$

This was a bit difficult, at least for me. (Perhaps it was easier for you.) With more complex chemical equations, you might be guessing and checking for a while. Unknown to many chemistry students, there is a way to balance chemical equations by using matrices, that does not involve any guessing.

2.4.2. Five Cool Examples

Here are five interesting chemical reactions that will serve as our examples.

(1) This reaction comes up in the smelting of ferrous oxide (iron ore) to extract the iron.

$$? \ Fe_2O_3 + ? \ CO \rightarrow ? \ Fe + ? \ CO_2$$

(2) When you put gasoline in your car, octane is one of the important chemicals. This is the reaction when your car burns octane in an abundance of oxygen. If you "rev" the engine up too high, other chemical reactions resulting in CO or even C can occur instead.

$$? \ C_8H_{18} + ? \ O_2 \rightarrow ? \ CO_2 + ? \ H_2O$$

(3) Here is an example of an acid-base reaction, a type of reaction fairly important in chemistry. Calcium hydroxide ($Ca(OH)_2$) has numerous uses, including the treatment of human sewage. Phosphoric acid (H_3PO_4) can be present in human sewage from sources as common as soda drinks. Here we see that calcium hydroxide will treat phosphoric acid and convert it to ordinary water and tricalcium phosphate. As it turns out, tricalcium phosphate is much more benign, as it occurs in animal bones and in cow's milk.

$$? \ Ca(OH)_2 + ? \ H_3PO_4 \rightarrow ? \ Ca_3(PO_4)_2 + ? \ H_2O$$

(4) In the previous example, we said that phosphoric acid (H_3PO_4) occurs in soda drinks. Where does it come from? It can be readily made from pure elemental phosphorous and hypochlorous acid ($HClO$) in the presence of water. Hydrochloric acid (HCl) is formed as an additional product. Here is the reaction:

$$? \ P_4 + ? \ HClO + ? \ H_2O \rightarrow ? \ H_3PO_4 + ? \ HCl$$

(5) This is one of the preferred methods of extracting gold from dilute rock samples. It is also horrendously toxic ($NaCN$ is sodium cyanide... which is as bad as it sounds!)

$$? \ NaCN + ? \ Au + ? \ O_2 + ? \ H_2O \rightarrow ? \ NaAu(CN)_2 + ? \ NaOH$$

2.4.3. The Slow Way: Deriving the Method

We're now going to slowly develop the matrix method of balancing a chemical equation. A shortcut approach will be given in the next section, but it will make no sense unless you read this derivation very carefully. We will take as our example the middle member of the five examples that we listed moments ago.

$$? \ Ca(OH)_2 + ? \ H_3PO_4 \rightarrow ? \ Ca_3(PO_4)_2 + ? \ H_2O$$

If I had x_1 molecules of $Ca(OH)_2$, x_2 molecules of H_3PO_4, x_3 molecules of $Ca_3(PO_4)_2$, and x_4 molecules of H_2O, then ...

- ... we would have x_1 atoms of calcium on the left, and $3x_3$ atoms of calcium on the right.
- ... we would have $2x_1+4x_2$ atoms of oxygen on the left, and $8x_3+x_4$ atoms of oxygen on the right.
- ... we would have $2x_1+3x_2$ atoms of hydrogen on the left, and $2x_4$ atoms of hydrogen on the right.
- ... we would have x_2 atoms of phosphorous on the left, and $2x_3$ atoms of phosphorous on the right.

Since, for each element, we need to have the same number of atoms on the left as on the right, then we have the following 4 equations in 4 variables:

$$\begin{aligned} x_1 &= 3x_3 \\ 2x_1 + 4x_2 &= 8x_3 + x_4 \\ 2x_1 + 3x_2 &= 2x_4 \\ x_2 &= 2x_3 \end{aligned}$$

The matrix equivalent to that system of equations would be

$$A = \left[\begin{array}{cccc|c} 1 & 0 & -3 & 0 & 0 \\ 2 & 4 & -8 & -1 & 0 \\ 2 & 3 & 0 & -2 & 0 \\ 0 & 1 & -2 & 0 & 0 \end{array}\right]$$

According to Sage, the RREF is

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

As you can see, we have a row of all zeros. We have a pivot for x_1, x_2, and x_3, thus those variables are determined. There is no pivot for x_4, so that variable is free. The "standard form" of the infinite set of solutions is written as follows:

$$\begin{aligned} x_1 &= (1/2)x_4 \\ x_2 &= (1/3)x_4 \\ x_3 &= (1/6)x_4 \\ x_4 &= x_4 \end{aligned}$$

However, it is customary only to consider integer values for the coefficients of a chemical reaction's equation. What do we do when we want integer-only solutions in a case like this? It is an extremely rare trick, not commonly taught but easy to learn. First, we take the lcm (least common multiple) of all the denominators of all the fractions. In this case, the lcm of $\{2, 3, 6\}$ is six. Second, we plug in $x_4 = 6n$ where n is a new dummy variable. Now we have

$$\begin{aligned} x_1 &= (1/2)6n = 3n \\ x_2 &= (1/3)6n = 2n \\ x_3 &= (1/6)6n = n \\ x_4 &= 6n \end{aligned}$$

For any integer n, the four values that you would get form an integer solution to the given equations. This includes nonsense values like $n = 0$ or negative values of n. (Or to be precise, the $n \leq 0$ cases are mathematically

valid solutions to the system of equations, but they are not chemically meaningful.) We want the simplest non-zero all-positive solution, so we should plug in $n = 1$. Now we have

$$x_1 = 3; \quad x_2 = 2; \quad x_3 = 1; \quad x_4 = 6$$

giving the final chemical reaction

$$3\ \mathrm{Ca(OH)_2} + 2\ \mathrm{H_3PO_4} \to 1\ \mathrm{Ca_3(PO_4)_2} + 6\ \mathrm{H_2O}$$

but we should note that chemists do not usually write the "1" in front of the third molecule.

Now we should check our work. As you can see, ...

- ...we have $3(1)+0 = 3$ atoms of calcium on the left, and $1(3)+0 = 3$ atoms of calcium on the right.
- ...we have $3(2) + 2(4) = 14$ atoms of oxygen on the left, and $1(8) + 6(1) = 14$ atoms of oxygen on the right.
- ...we have $3(2) + 2(3) = 12$ atoms of hydrogen on the left, and $0 + 6(2) = 12$ atoms of hydrogen on the right.
- ...we have $0 + 2(1) = 2$ atoms of phosphorous on the left, and $1(2) + 0 = 2$ atoms of phosphorous on the right.
- All of those are exact matches, therefore we declare success!

2.4.4. Balancing the Quick Way

That was a really excellent but slow process. Now we might want to balance chemical equations faster. The key to remember is "atoms are rows, molecules are columns." A hint to remembering that is to observe that the word "row" is shorter than the word "column," and similarly the word "atom" is shorter than the word "molecule." Pairing short-to-short and long-to-long, it is natural that atoms correspond to rows and molecules correspond to columns.

The atoms can be encoded in any order. Let's try the second example, the combustion of octane.

$$?\ \mathrm{C_8H_{18}} + ?\ \mathrm{O_2} \to ?\ \mathrm{CO_2} + ?\ \mathrm{H_2O}$$

Let's go one atom at a time. We're going to ask each molecule how many of the particular atom it has. For the reagents (the left-hand side of the arrow), we will use positive integers; for the products, (the right-hand side of the arrow), we will use negative integers.

Carbon: The four molecules have 8, 0, 1, and 0 atoms of carbon, thus we write `8, 0, -1, 0`.

Hydrogen: The four molecules have 18, 0, 0, and 2 atoms of hydrogen, thus we write `18, 0, 0, -2`.

Oxygen: The four molecules have 0, 2, 2, and 1 atoms of oxygen, thus we write `0, 2, -2, -1`.

Now we define the matrix, and ask for its RREF, in the notation that we learned about on Page 23. However, we must be careful to add the column

of zeros as the rightmost column of the matrix, signifying the $= 0$ part of the equations.

```
B= matrix([ [8, 0, -1, 0, 0], [18, 0, 0, -2, 0], [0, 2, -2, -1, 0]])

print "Question:"
print B
print "Answer:"
print B.rref()
```

Now we have the output

```
Question:
[ 8  0 -1  0  0]
[18  0  0 -2  0]
[ 0  2 -2 -1  0]
Answer:
[     1      0      0   -1/9      0]
[     0      1      0 -25/18      0]
[     0      0      1   -8/9      0]
```

Quickly, we verify that the matrix that we entered was the matrix that we had desired to enter. Next, we take the lcm of the denominators in the interesting column. The lcm of $\{9, 18, 9\}$ is clearly 18. For the first three coefficients, we multiply the entries in informative column by the negative lcm (in this case, -18), and obtain 2, 25, and 16. For the fourth coefficient, it is just the lcm itself, namely 18. We conclude with

$$2\ C_8H_{18} + 25\ O_2 \to 16\ CO_2 + 18\ H_2O$$

but now we have to check it.

- We have $2(8) + 0 = 16$ atoms of carbon on the left, and $16(1) + 0 = 16$ atoms of carbon on the right.
- We have $2(18) + 0 = 36$ atoms of hydrogen on the left, and $0 + 18(2) = 36$ atoms of hydrogen on the right.
- We have $0+25(2) = 50$ atoms of oxygen on the left, and $16(2)+18 = 50$ atoms of oxygen on the right.
- Since the numbers match, we declare victory!

Last but not least, it would be good to note that chemists have shortcuts that they use which simplify matters significantly, but which require chemical intuition. For example, PO_4 and OH are polyatomic ions, which means that you can "pretend" they are individual atoms. This can remove one variable from the linear system of equations, which is significant.

Your Challenge:

Now try the first, fourth, and fifth examples on your own. Good luck, and don't forget to check your work!

2.5. Physics: Ballistic Projectiles

Ballistic projectile motion problems are a staple in calculus classes as well as physics classes. Here we assume that the reader has knowledge of *Calculus I* and *Physics I*, or the high school equivalents. We're going to explore two relatively easy problems—where there is no need to use Sage—and then one that is a bit more difficult—where Sage or some other computational tool would be required.

For a wide class of simple and intermediate problems, the following seven-step technique often completely solves the problem.

(1) Read the problem very carefully, and write down a sum of forces equation—often with the aid of a free-body diagram. This, in turn, produces an acceleration function using $\Sigma F = ma$.
(2) Integrate the acceleration function to get the velocity function, which unfortunately will have a $+C$.
(3) Use some information from the problem to compute the value of C.
(4) Integrate the velocity function to get the altitude function, which unfortunately will have a (probably different) $+C$.
(5) Again, use some information from the problem to compute the value of C.
(6) Check that your three functions are consistent with all the given data (by plugging in values) and with each other (by taking derivatives).

Note: At this point, you have all three functions explicitly and can answer most any inquiry about the projectile.

(7) Reread the question and see what is actually asked for.

Throughout this project, $y(t)$ will represent the altitude of a projectile, and $y'(t) = v(t)$ its velocity. Likewise, $y''(t) = v'(t) = a(t)$ is the projectile's acceleration. We will ignore air resistance in all cases. Due to the perils of rounding error, it is unfortunately necessary that we use $g = 9.80665$ m/s^2 throughout this problem—we really do need all six significant figures.

Before we start, it should be stated that there are two ways of approaching these sorts of problems. Typically, applied mathematicians use lots of digits of precision midway, and only round at the end, to reflect the uncertainty in the inputs. However, they do not usually have the units inside of equations attached to all of the coefficients and constants. In contrast to this, physicists use units inside of equations (!) attached to all of the coefficients and constants. Furthermore, in the first-year university courses, some textbooks round off numbers stage-by-stage in the middle steps of the computations, to reflect the limited precision of the input data. Since I am accustomed to the applied mathematician's technique, I use it here, so as to minimize the chance of a typographical mistake.

You can do these next two warm-up examples with a pencil and a scientific calculator. Then we will proceed to a problem where we need Sage.

2.5.1. Our First Example of Ballistic Trajectories

Suppose an artillery piece is located on a hill, such that the muzzle is 25 m above the surrounding countryside. It fires its projectile at a muzzle velocity of 300 m/s, at an angle of 30° above the horizontal. How far does the projectile go?

First, we have to construct an acceleration function. The only force on the projectile (mid-flight) is the weight of the projectile due to the earth's gravity. We write

$$\begin{aligned} \Sigma F &= ma(t) \\ -mg &= ma(t) \\ -g &= a(t) \\ a(t) &= -9.80665 \end{aligned}$$

Second, we integrate to find $v(t)$ with a +C.

$$\begin{aligned} \int a(t)\ dt &= \int a(t)\ dt \\ \int v'(t)\ dt &= \int -9.80665\ dt \\ v(t) &= -9.80665t + C \end{aligned}$$

Third, we know that $v(0)$ is the vertical component of the muzzle velocity. That's computed with

$$(300)(\cos 30^\circ) = (300)(0.866025) = 259.807$$

and therefore

$$\begin{aligned} v(t) &= -9.80665t + C \\ v(0) &= -9.80665(0) + C \\ 259.807 &= C \end{aligned}$$

Now we know that $v(t) = -9.80665t + 259.807$, at least during flight. Fourth, we integrate to find $y(t)$ with a +C.

$$\begin{aligned} \int v(t)\ dt &= \int v(t)\ dt \\ \int y'(t)\ dt &= \int (-9.80665t + 259.807)\ dt \\ y(t) &= -\frac{1}{2}(9.80665)t^2 + 259.807t + C \\ y(t) &= -4.90332t^2 + 259.807t + C \end{aligned}$$

Fifth, we plug in the fact that $y(0)$ is the altitude of the hill, or 25 m.

$$\begin{aligned} y(t) &= -4.90332t^2 + 259.807t + C \\ y(0) &= -4.90332(0)^2 + 259.807(0) + C \\ 25 &= C \end{aligned}$$

Finally, we have

$$y(t) = -4.90332t^2 + 259.807t + 25$$

and we can begin the most important part of the problem: checking our work.

- We can check that $y'(t) = -9.80664t + 259.807$, which matches our $v(t)$ except for rounding error.
- We can check that $y''(t) = -9.80664$, which matches our $a(t)$ except for rounding error.
- We can check that $y'(0) = v(0) = 259.807$, as desired. (That's the vertical component of the muzzle velocity.)
- We can check that $y(0) = 25$, as desired. That's the altitude of the artillery piece's muzzle.

Now, we should compute what the question actually wants to know. It wants to know the range of the projectile (how far it goes) so we have to calculate the time of flight and multiply that by the horizontal velocity. To find the time of flight, we just have to find out when the projectile hits the ground (when flight ends) because flight begins when $t = 0$. Of course, when the projectile hits the ground, the altitude is zero. Therefore, we set $y(t) = 0$, and use the quadratic formula.

$$\begin{aligned} 0 &= y(t) \\ 0 &= -4.90332t^2 + 259.807t + 25 \\ t &= \frac{-259.807 \pm \sqrt{(259.807)^2 - 4(-4.90332)(25)}}{2(-4.90332)} \\ t &= -0.0960512 \text{ or } 53.0820 \end{aligned}$$

We check our work with $y(53.0820) = 0.0174218$ m, a difference of 17 mm, which is not a problem, but which is obviously mere rounding error. Next, we need the horizontal velocity

$$(300)(\sin 30^\circ) = (300)(1/2) = 150$$

and finally the range of the projectile is

$$(150)(53.0820) = 7962.30$$

Since a mile is 1609 meters, a range of 7962.30 m seems plausible. After all, the purpose of artillery is to strike targets several miles away. We should realize, however, that our altitude of the hill being 25 m is a measurement accurate probably to the nearest meter or half meter, and not to six significant digits. The muzzle velocity is probably also only known to the nearest

1%. Therefore, it is better to report the answer to two significant figures, and reply that the projectile's range is 7900–8000 meters.

2.5.2. Our Second Example of Ballistic Trajectories

Now we're going to explore an air-defense problem. We're going to simulate the computer of an air-defense radar system that is tracking incoming warheads. This is not an unrealistic problem. During the First Gulf War (1991), Iraq fired SS-1 "Scud" missiles over long distances to Israel and Saudi Arabia. Some could be shot down during flight, by surface-to-air missiles, but there were many incoming Scuds. It is useful to compute the point of impact—after all, if a warhead is heading toward the empty desert, then that's fine, but if a warhead is heading toward a shopping mall, then that's an emergency. Just as this book was going to the publisher in July of 2014, similar computations were being done by the Israeli air-defense system "Iron Dome," so rest assured that these techniques are rather relevant.

To keep the numbers and the model simple, we'll work with a problem about artillery shells, and not about regional ballistic missiles.

A projectile has been detected in the vicinity of the air-defense battery for which our computer has responsibility. The radar's computer has defined its coordinate system so that the location of the radar is $(0, 0, 0)$. A radar snapshot has been taken of the incoming projectile, reporting its position and velocity. The y direction is straight up, the x direction is due east, and the z direction is due south. The radar has reported that the projectile is at a position of $(12,000; 5000; 7000)$ meters, with a velocity of $(-60, -240, 80)$ in m/s. The location of impact must first be computed. For simplicity, let $t = 0$ be the time of the snapshot, and not the time of launch.

First, we should use the seven-step method to find $y(t)$, the altitude function. I'll allow you to do that yourself, so that you can practice. However, I'll give you the answers after each step, so that you can check your work.

(1) Step One: $a(t) = -9.80665$
(2) Step Two: $v(t) = -9.80665t + C$
(3) Step Three: $v(t) = -9.80665t - 240$
(4) Step Four: $y(t) = -4.90332t^2 - 240t + C$
(5) Step Five: $y(t) = -4.90332t^2 - 240t + 5000$
(6) Step Six:
    - The first derivative is $y'(t) = -9.80664t - 240$, matching $v(t)$.
    - The second derivative is $y''(t) = -9.80664$, matching $y(t)$.
    - The altitude at $t = 0$ is $y(0) = 5000$, as desired.
    - The altitude at $t = 0$ is $y'(0) = -240$, as desired.
(7) Step Seven: We need to find the impact time, just like last problem. However, this does not happen for every problem. In any case, the time of impact is somewhat soon, at $t = 15.7593$ seconds.

(8) Step Seven, Continued: At a rate of $\dot{x} = -60$ m/s, in 2.00148, the projectile will have moved -945.558 in the x-direction; at a rate of $\dot{z} = +80$, the projectile will have moved 1260.74 in the z-direction. Of course, at impact, $y = 0$.

(9) Therefore the point of impact is $(-11,054.4; 0; 8260.74)$.

Note: The symbols $\dot{x}$ and $\dot{z}$ are just abbreviations for dx/dt and dz/dt.

2.5.3. Counter-Battery Fire:

In real life, one of the fun things that can be done is to mathematically trace the parabola of flight backward to detect the launch point. Once the point of launch is known, clearly there is an artillery piece there, and therefore it would be prudent to heavily bombard that location to destroy the enemy artillery battery. This technique, called "counter-battery fire," was developed in the First World War, and has been a mainstay of modern warfare since that time. Another reason the technique is fun is that you use "the wrong value" of t when solving $y(t) = 0$. Namely, you use the root with $t < 0$, something which almost never happens in projectile motion problems.

The time of launch is when $y(t) = 0$ but $t < 0$. (Actually, we're tacitly assuming that the altitude of the launcher, the observer, and the impact are all equal or nearly equal. However, if they were not equal, the adjustments would be straightforward.) The quadratic equation reveals to us that in this case, $t = -64.7057$ seconds. This sounds reasonable, as it makes the total time of flight 80.4650 seconds, being a bit more than one minute.

The coordinates of the firing howitzer can be obtained by computing

$$\begin{aligned} (-60)(-64.7057) + 12,000 &= 15,882.3 \\ (+80)(-64.7057) + 7000 &= 12,176.4 \end{aligned}$$

Therefore, we report that the enemy artillery battery is at the following coordinates:

$$(15,882.3; \quad 0; \quad 12,176.4)$$

The Truth about Air Resistance:

Of course, we've neglected wind and air resistance—in real life, it would be better to include them. You're probably curious how this is actually done. What happens is that time is divided into tiny intervals, around 10^{-4} or 10^{-5} seconds, and during those time slices, one pretends that the effect of wind and air resistance is a constant function or a linear function. About 10^6 or 10^7 time slices represents the entire flight path of the projectile. Using techniques that you will learn in the first parts of Chapter 5, it is extremely easy to write a computer program which repeatedly steps forward in time by those tiny time slices.

The modification by a constant function or linear function is going to only slightly elevate the complexity of the calculation for each time slice. However, because the time slices are so tiny, the accuracy of the overall

computation can be very high. The general concept we are discussing here is called "numerical simulation."

2.5.4. Your Challenge: A Multi-Stage Rocket

Now that you have practiced the techniques of projectile motion, let's consider the test flight of a rocket. In this case, the rocket malfunctions and crashes, and we are going to model its flight path.

During normal operations, the engine provides a constant force of thrust of 1.27 meganewtons. However, eleven seconds into flight, the fuel pump control computer malfunctioned and the flow of fuel to the engine was slowly throttled to zero. Of course, the rocket crashed. A good model of the thrust of the engine after the malfunction would be

$$f(t) = 1.27e^{(t-11)/2} \text{ for all } t > 11$$

again measured in meganewtons. It will be useful to know that the mass of the rocket at launch was 107,000 kilograms.

We have two simplifying assumptions. While fuel is clearly being burned during the first eleven seconds, it is okay to pretend as though the mass is constant for the entire problem. (That's because only a tiny fraction of the fuel was burned during the flight.) Second, the rocket does not get very high at all, and therefore it is okay to pretend $g = 9.80665$ m/s^2 is a constant. In reality, it does change when you reach higher and higher altitudes, but a projectile has to get relatively high before that matters too much.

Let's begin. First, find the altitude function when the engine was working normally—namely $0 < t < 11$. You can assume that the rocket was sitting on the launch pad, at $y = 0$, for a long time prior to launch. Once you have this, compute the altitude, velocity, and acceleration 11 seconds after launch—the moment that the malfunction occurred.

Next, using the function $f(t)$ above, compute a new acceleration function, a new velocity function, and a new position function, for use after the malfunction. That will enable us to find the time of impact.

There are ten deliverable items for this project:

- First, second, and third, provide the acceleration, velocity, and position functions before the malfunction.
- Fourth, fifth, and sixth, provide the acceleration, velocity, and position functions after the malfunction.
- Seventh, provide the time of impact.
- Eight, provide a single graph showing the altitude for the entire flight, from launch to impact.
- Ninth, provide a single graph showing the velocity for the entire flight, from launch to impact.
- Tenth, provide the maximum height achieved.

To help you understand if you've done the problem correctly, the two graphs are given in Figure 2.

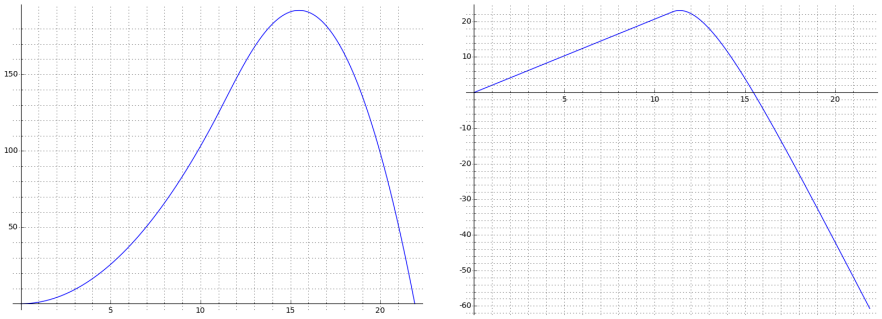


FIGURE 2. The Graphs for the Physics Project

2.6. Cryptology: Pollard's $p-1$ Attack on RSA

The RSA Cryptosystem is easily the most famous of cryptographic systems in use today. Nearly every E-commerce transaction on the internet uses TLS (Transport Layer Security) or SSL (the Secure Sockets Layer) and therefore uses RSA to exchange block-cipher keys. The security of RSA is based upon the difficulty of factoring the product of two (enormous) prime numbers.

This project is one of several ways of attacking the RSA Cryptosystem. The project assumes familiarity with RSA and a bit of number theory. If the reader is unfamiliar with RSA, then this project cannot be attempted. However, I would encourage such a reader to learn RSA. It is relatively easy to learn, at least compared to other mathematical concepts, and it provides a window into the practical applications of otherwise "pure" topics such as prime numbers and modular arithmetic.

This project closely follows the notation given in *Cryptography and Coding Theory*, Second Edition, by Wade Trappe and Lawrence Washington, published by Pearson in 2005. See Ch 6.4 of that textbook. The original method was discovered by John Pollard in 1974.

Alternatively, Minh Van Nguyen has written a tutorial for Sage and cryptography that uses RSA as its example. The title is "Number Theory and the RSA Public Key Cryptosystem" and you can find that tutorial at

`http://www.sagemath.org/doc/thematic_tutorials/numtheory_rsa.html`

Last but not least, this project requires some knowledge of `for` loops, as covered in Ch. 5 of this book.

2.6.1. Background: B-Smooth Numbers and $B!$

Number theorists will often talk about certain positive integers as being "smooth." This is a vague term meaning that the number has no large prime factors, but rather that the prime factorization is entirely composed of small primes. More precisely, for any even number k, we say that a number is k-smooth if all of its prime factors are less than k. As example, 15, 10, 100 and 2^{5000} are all 6-smooth, but 14 is not, because $14 = 2 \times 7$ and

7 is larger than 6. As another example, 14 and 1400 are 8-smooth, but 11 is not 8-smooth. This notion can be interesting in several branches of number theory, but it is especially of interest in cryptography. This is the standard way of defining what it means for a number to be "smooth."

As it comes to pass, an RSA public key can easily be broken (by factoring its N), if either $p-1$ or $q-1$ are smooth. That's a rather weird statement, of course. First of all, p and q are not public information. However, that's no barrier to this attack, because it is only required that $p-1$ or $q-1$ be smooth—we do not need to know $p-1$ nor $q-1$, or even which of the two primes is "one plus a smooth number."

Another oddity is that we're going to use a non-standard definition of smoothness. We're going to pick a large number B and ask if either p or q divides $B!$. Shortly, we'll use $B = 10,000$, but for now, we'll use $B = 100$. Let's explore how this usage of 100! compares to a number being 100-smooth.

The only way that a number z fails to be 100-smooth is if z has a prime in its prime factorization that is greater than 100, such as perhaps 101 or 103. Since 101 and 103 are prime, and since both are greater than 100, we know that neither 101 nor 103 will divide any of the numbers from 1 to 100. Therefore, by the famous number theory lemma that $p|ab$ if and only if $p|a$ or $p|b$, we can conclude that 101 does not divide 100!, and likewise 103 does not divide 100!. Likewise, any multiple of 101 or 103 will also not divide 100!. The same goes for any prime larger, such as 107 and so forth. Therefore, any number which has a prime in its factorization that is greater than 100, will definitely not divide 100!.

However, we might be interested in the converse. If a number y does not divide 100!, is it always the case that y is not 100-smooth? Sadly, here we will encounter some exceptions. Using the commands

```
factor( factorial( 100 ) )
```

we learn that

$$\begin{aligned} 100! &= (2)^{97}(3)^{48}(5)^{24}(7)^{16}(11)^{9}(13)^{7}(17)^{5}(19)^{5}(23)^{4}(29)^{3}(31)^{3}(37)^{2}\cdots \\ &\quad \cdots(41)^{2}(43)^{2}(47)^{2}(53)(59)(61)(67)(71)(73)(79)(83)(89)(97) \end{aligned}$$

Therefore, you can see that 2^{98}, 3^{49}, and 5^{25} will not divide 100!. However, though those three numbers are all 6-smooth, so they are surely 100-smooth. So we have now located three examples of 100-smooth numbers that do not divide 100!.

In fact, we've now demonstrated that requiring numbers to be divisors of 100! is *a stricter standard* than requiring them to be 100-smooth. In general, the set of numbers dividing $B!$ is a subset of the numbers that are B-smooth. Overall though, if you consider numbers less than $B!$, the exceptions are few and far between, and the concepts are roughly equivalent.

2.6.2. The Theory Behind the Attack

We need one more theorem from Number Theory. For any prime number p, and for any $a \not\equiv 0 \mod p$, Fermat's Little Theorem says that

$$a^p \equiv a \mod p \qquad \text{or equivalently} \qquad a^{p-1} \equiv 1 \mod p$$

Suppose $p-1$ divides $B!$. That means that there is some k such that $(k)(p-1) = B!$. Then we can conclude

$$2^{B!} = 2^{(k)(p-1)} = (2^{p-1})^k \equiv 1^k \equiv 1 \mod p$$

Because $2^{B!} \equiv 1 \mod p$, it must be the case that

$$(2^{B!} - 1) \equiv 0 \mod p$$

or more plainly, that $c = (2^{B!} - 1)$ is a multiple of p. This key fact, that c is a multiple of p, is the heart of the entire idea.

In fact, on further thought, it must be the case that both c is a multiple of p in the ordinary integers, and that the image of $c \mod N$ is a multiple of p in "the integers mod N." Most readers will take this rather technical point on faith, and can skip to the start of the next paragraph. For those who want absolute rigor, observe that if c is a multiple of p then $c = jp$, for some integer j. We have no reason to believe that j is divisible by q, but let's divide by j anyway, getting quotient j_1 and remainder j_2. This means that $j = qj_1 + j_2$ and thus

$$c = jp = (qj_1 + j_2)p = pqj_1 + pj_2 = Nj_1 + pj_2 \equiv 0 + pj_2 \mod N$$

which allows us to conclude that the image of $c \mod N$ is just pj_2, clearly a multiple of p.

What have we gained by this? Well, since $N = pq$, we know that N is a multiple of p as well. Since both c and N are multiples of p, then $\gcd(c, N)$ is also a multiple of p. That gcd can be only one of two values, actually—that gcd is either p or N.

Therefore, we first compute $c = 2^{B!} - 1 \mod N$, which is no minor task, and then compute the value of $\gcd(c, N)$. If we get something between 1 and N, we know it must be p, and ordinary division will provide us with $q = N/p$. We have factored N and we have broken this person's public key.

On the other hand, if we get that the gcd is N, we are unlucky and have[3] failed. The gcd will be N (resulting in a failure of the attack) if and only if q also divides c. That will essentially only occur if both $p-1$ and $q-1$ divide $B!$.

As it comes to pass, having one of *either* $p-1$ or $q-1$ divide $B!$ is somewhat rare. However, having *both* $p-1$ and $q-1$ dividing $B!$ is extremely rare. More precisely, p and q are supposed to be generated randomly, and

[3]If this ever happens to you in practice, try again with a smaller B. If the largest prime in the prime factorization of $p-1$ is f_1 and then largest prime in the prime factorization of $q-1$ is f_2, then choosing a B between f_1 and f_2, presumably by guess-and-check, will result in a successful outcome—namely $\gcd(c, N) = p$.

therefore the probability of $p-1$ dividing $B!$ is independent of the probability of $q-1$ dividing $B!$. The probability of both of them dividing $B!$ is therefore *the square of the probability* of one of them happening to divide $B!$.

Accordingly, we fully expect that the attack will work for any N, so long as either $p-1$ or $q-1$ divides $B!$.

2.6.3. Computing $2^{(B!)} \mod N$

Let's reflect upon the fact that computing

$$2^{(10,000!)} \mod N$$

is not a task that we are asked to do very often. Therefore it would be unwise to dive into the problem without some thought at first. We might ask ourselves, exactly how large is $10,000!$ going to be? Or, more precisely, how many digits will it have?

It turns out this question is easy to answer if we recall Stirling's approximation:

$$B! \approx B^B e^{-B} \sqrt{2\pi B}$$

and proceed by taking the common logarithm

$$\begin{aligned}
\log_{10}(B!) &\approx \log_{10}\left(B^B\right)\left(e^{-B}\right)\left(\sqrt{2\pi B}\right) \\
&\approx \left(\log_{10} B^B\right) + \left(\log_{10} e^{-B}\right) + \left(\log_{10}\sqrt{2\pi B}\right) \\
&\approx (B\log_{10} B) - (B\log_{10} e) + \frac{1}{2}\left(\log_{10} 2\pi B\right) \\
&\approx B\left(\log_{10} B - \log_{10} e\right) + \frac{1}{2}\left(\log_{10} 2\pi\right) + \frac{1}{2}\left(\log_{10} B\right) \\
&\approx B\left(\log_{10} B - 0.434294481\cdots\right) + 0.399089934\cdots + \frac{1}{2}\log_{10} B \\
&\quad \ldots \text{or in our case of } B = 10,000 \text{ and } \log_{10} B = 4\ldots \\
&\approx 10,000\left(4 - 0.434294481\cdots\right) + 0.399089934\cdots + \frac{1}{2}(4) \\
&\approx 35,657.0551\cdots + 2.39908993\cdots \\
&\approx 35,659.4542\cdots
\end{aligned}$$

Having come to realize that $10,000!$ is a number that has 36,660 digits (approximately) we probably should not compute it and then ask for $2^{(10,000!)}$. That might take a very long time to compute. We have to find a different approach.

Consider computing

$$\begin{aligned} c_1 &= 2^1 \mod N \\ c_2 &= (c_1)^2 \mod N \\ c_3 &= (c_2)^3 \mod N \\ c_4 &= (c_3)^4 \mod N \\ c_5 &= (c_4)^5 \mod N \\ c_6 &= (c_5)^6 \mod N \\ \vdots &\ \vdots\ \ \vdots \end{aligned}$$

Then we would have

$$c_6 \equiv (c_5)^6 \equiv ((c_4)^5)^6 \equiv (((c_3)^4)^5)^6 \equiv ((((c_2)^3)^4)^5)^6 \equiv (((((c_1)^2)^3)^4)^5)^6 \equiv 2^{(1)(2)(3)(4)(5)(6)}$$

but observe, that's just what we wanted to compute, namely

$$2^{(1)(2)(3)(4)(5)(6)} \equiv 2^{(6!)}$$

In a similar manner, we can see that $c_B \equiv 2^{(B!)} \mod N$.

2.6.4. Your Challenge: Make All This Happen in Sage

Your task is an easy one. Using the strategy above, compute

$$c = 2^{(10{,}000!)} \mod N$$

Next, compute $p = \gcd(c, N)$, followed by $q = N/p$ and check your work by multiplying p and q back together again. Here are seven test cases for you:

3571770308958050087813192668766411035818396781514951351291330670101 27

5000272002251811540446579586816039346308785565641862331905860763123733

1112930940347693048841670514581743208135955104481382748661520020673659 27

1891149894297520829262854575513696427873817900392608023074521104903045 47

2507336544824685689216200428668597188529200280260765471779747582391091 77

2015573899005400956135592195412995405224052593293997368248582528763765213110530067105771630570340 93

8620215476436315823969982122087229142882586442347913079505829164427472220397956094177419322783171 21

By the way, you should be able to cut-and-paste these numbers into Sage from the `pdf` file of this book, available on my[4] webpage. You do not need to retype these enormous numbers!

2.6.5. Safeguards against Pollard's Factorial Attack

As you can see, RSA public keys can be broken (that is to say, N can be factored) if either $p-1$ or $q-1$ are smooth. A smooth number (for example, a 6-smooth or 100-smooth number) has a rather large number of prime factors which each contribute a small magnitude toward the final product. For example,

$$10^{100} = 2^{100}5^{100}$$

[4] `http://www.gregorybard.com/`

is a very smooth number, with 200 primes in its prime factorization (counting multiplicities) and none contribute more than a paltry 5 to final, enormous, product. What is the opposite extreme to this?

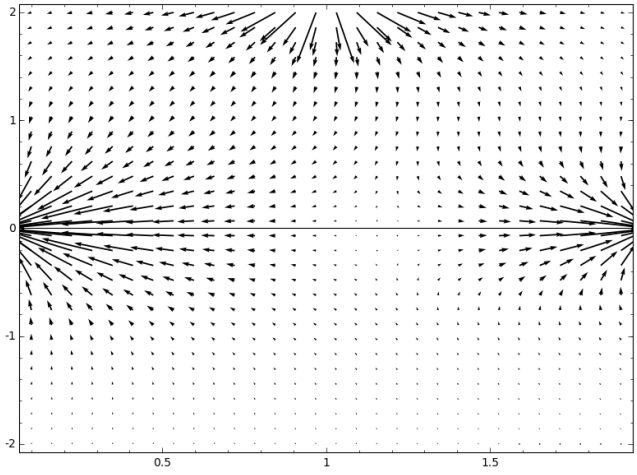
The other extreme would be a prime number. There is only one prime factor, and it makes all the contribution to the magnitude by itself. Of course, p and q are odd and thus neither $p-1$ nor $q-1$ can be prime. However, they could be 2 times some prime number. Prime numbers p where both p and $(p-1)/2$ are prime simultaneously are called "safe primes." The almost identical notion, where both p and $2p+1$ are prime, are called "Sophie Germain" primes, named for Marie-Sophie Germain (1776–1831).

To make Pollard's attack totally infeasible, many RSA implementations require both primes to be "safe primes." For these numbers, B would have to be $(p-1)/2$ or greater. Even if p is only 200 bits long, which is far too small, then $(p-1)/2$ will be 199 bits long, and that would make it computationally infeasible to compute $2^{B!}$. The sun would run out of fuel, turning into a red giant[5] long before such a computation could terminate.

2.7. Mini-Project on Electric Field Vector Plots

Take a moment to read Section 3.7: Vector Field Plots, beginning on Page 114, if you haven't already read it. In this mini-project, you're going to be asked to modify the model that I used in the subsection "An Application from Physics" to generate the vector field plot of three electric charges. We'll be adding a fourth electric charge, with $q_3 = -1$, and it is glued to the point $(1, 2.25)$.

Your task is to modify my code to accommodate this fourth electric charge, and then produce the vector field plot. You should obtain the following image:



[5]This will happen roughly 5 billion years from now.

Chapter 3

Advanced Plotting Techniques

One of the most enjoyable uses of Sage is to produce colorful graphic images, whether for teaching or for publication. Sage has an enormous variety of plotting and graphing utilities. The basics of plotting were first introduced in Section 1.4 on Page 8. If you haven't read that yet, you'll want to go read that now. Almost everything else you would want to plot is covered here in this chapter. The exception are certainly highly colorful or 3D plots which can only appear on a computer screen and not in a printed book. Those plots are discussed in the electronic-only online appendix to this book "Plotting in Color, in 3D, and Animations," available on my webpage `www.gregorybard.com` for downloading.

Moreover, because scatter plots are essentially only used while making regressions, the technique for scatter plots is discussed in Section 4.9 on Page 157, for simplicity. Similarly, slope field plots are only used (so far as I am aware) in the study of first-order differential equations, and therefore those procedures are discussed in Section 4.22.3 on Page 197 for better continuity of the exposition.

3.1. Annotating Graphs for Clarity

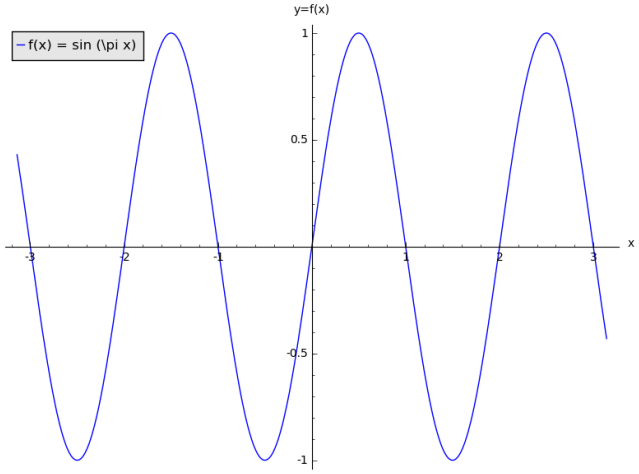
In this section, we have a collection of techniques for making 2D plots look attractive. They include labeling axes, add gridlines, adding a frame, marking a point with a big dot, drawing an arrow on a plot, inserting text into a plot, shading plots (for integrals), and using dotted or dashed lines.

3.1.1. Labeling the Axes of Graphs

Sometimes it is very useful to label the axes of a graph, directly on the graph. This is particularly handy when there is a risk of misunderstanding the units involved, or in a scientific application where many functions are discussed. In any case, the following code accomplishes that objective.

```
p= plot(sin(pi*x), -pi, pi, legend_label='f(x) = sin (pi x)' )
p.axes_labels(['x',  'y=f(x)'])
p.show()
```

The above code, provided by Martin Albrecht and Keith Wojciechowski, produces the following image:



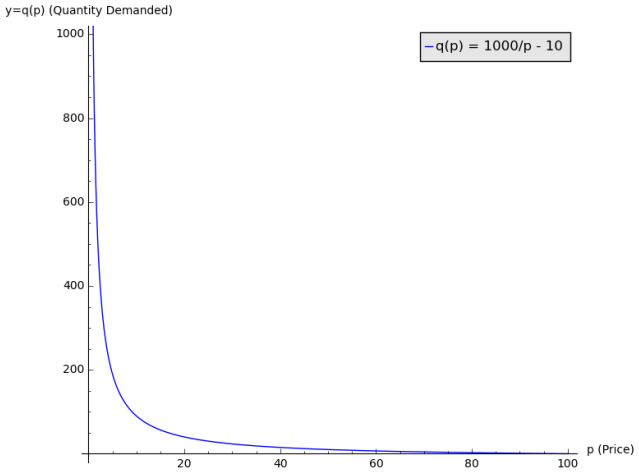
Here is another example from economics. The demand curve

$$q(p) = \frac{1000}{p} - 10$$

relates the quantity demanded by the market of some product to its price. The following is some code to plot the above curve, with informative labels.

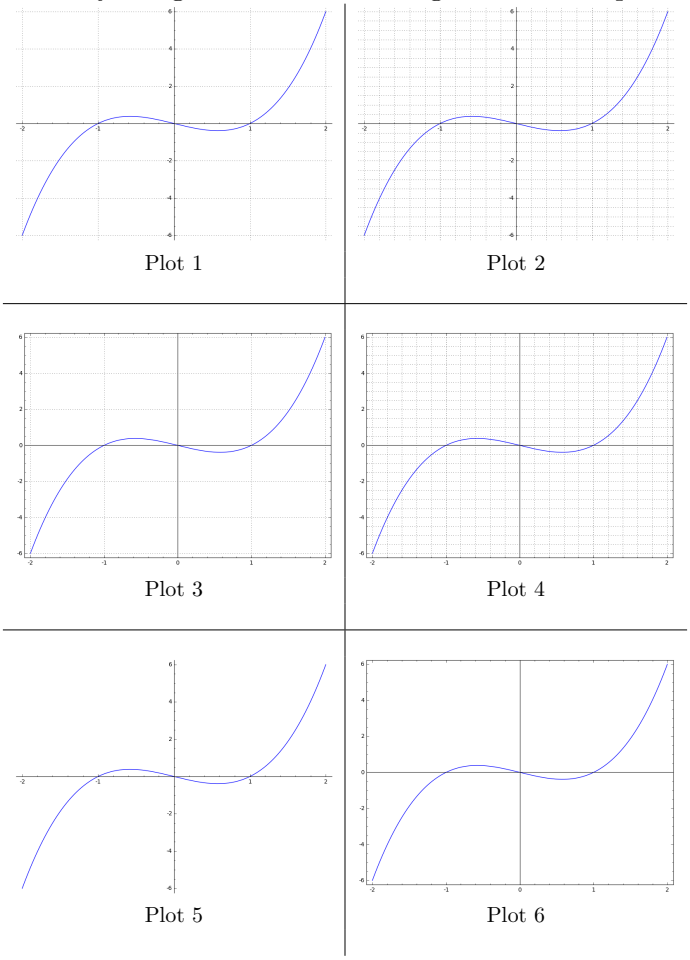
```
p = plot(1000/x - 10, 0, 100, ymax = 1000,
          legend_label=  'q(p) = 1000/p - 10' )
p.axes_labels(['p (Price)',  'y=q(p) (Quantity Demanded)'])
p.show()
```

The image produced by that code is below.



3.1.2. Grids and Graphing Calculator-Style Graphs

Sometimes in a plot, it is nice to show the grid lines, as if the graph were drawn on a piece of graph paper. There are two options for this, namely `frame`, which can be `True` or `False`, and `gridlines`, which can be `True`, `False`, or `"minor"`. Therefore, there are actually six different choices of layout. They are given in the following collection of plots.



Plot 1 Plot 2

Plot 3 Plot 4

Plot 5 Plot 6

Here is the code that generated each of those plots.

Plot 1:

```
plot(x^3-x, -2, 2, gridlines=True, frame=False)
```

Plot 2:

```
plot(x^3-x, -2, 2, gridlines='minor', frame=True)
```

Plot 3:

```
plot(x^3-x, -2, 2, gridlines=True, frame=True)
```

Plot 4:

```
plot(x^3-x, -2, 2, gridlines='minor', frame=True)
```

Plot 5:

```
plot(x^3-x, -2, 2, gridlines=False, frame=False)
```

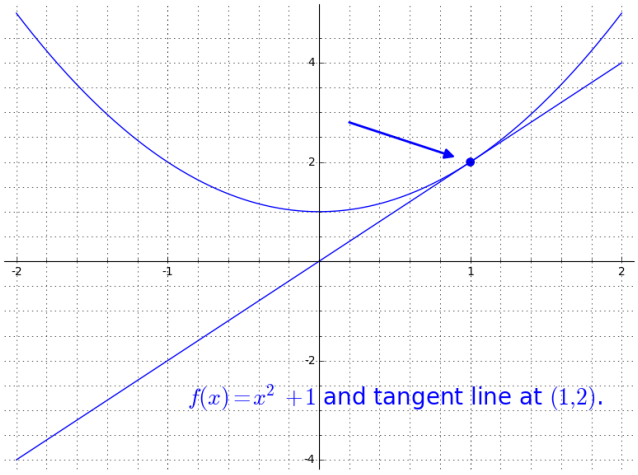
Plot 6:

```
plot(x^3-x, -2, 2, gridlines=False, frame=True)
```

3.1.3. Adding Arrows and Text to Label Features

Sage has some nice techniques for making graphs easier to understand. We've already seen how the `point` command can be used to draw a large dot on a plot in order to draw the viewers attention. When needed, the `arrow` command can be used to draw a rather large and non-subtle arrow. The arrow is defined by two points: its tail and its head, with the tail coordinates given first and the head coordinates given second. The syntax is shown below. Also, the `text` command can be used to add some words directly to plot. It has two parameters: the message to be added and the center of the desired words.

```
tangent_line = plot( 2*x, -2, 2, gridlines='minor' )
curve = plot( x^2+1, -2, 2 )
big_dot = point( (1,2), size=60 )
the_arrow = arrow(  (0.2, 2.8),   (0.9, 2.1) )
the_words = text( '$f(x) = x^2 + 1$ and tangent line at $(1,2)$.',
                 (0.5, -2.75), fontsize=20  )
big_image = tangent_line + curve + big_dot + the_arrow + the_words
big_image.show()
```



Last but not least, it is worth mentioning that the `text` command knows the mathematical document preparation language "LaTeX." If you don't know LaTeX, then you can type instead

```
the_words = text( 'f(x) = x^2 + 1 and tangent line.',
          (0.5, -2.75), fontsize=20 )
```

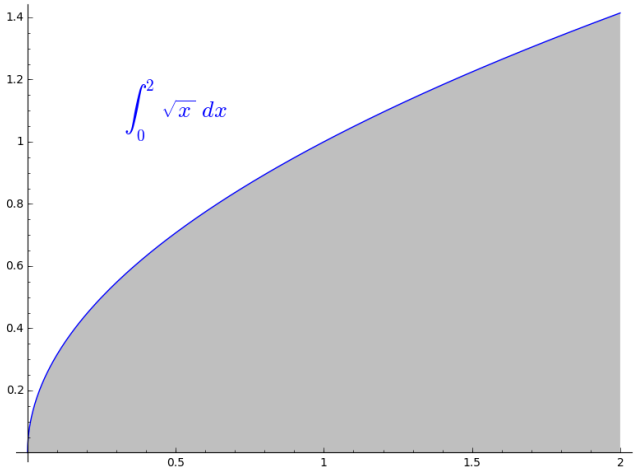
and have almost the same effect.

3.1.4. Graphing an Integral

Here is another example to show the power of LaTeX, as well as how to graph with shading, such as when teaching about integrals.

```
P1= text('$\int_0^2 \sqrt{x} \, dx$', (0.5, 1.1), fontsize=20)
P2= plot( sqrt(x), 0, 2, fill=True)
P= P1 + P2
P.show()
```

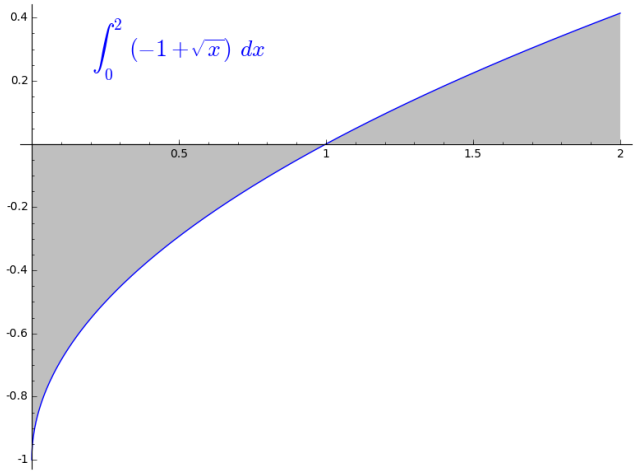
The following image is produced:



The fill command is excellent for working with integrals, even of functions that cross the x-axis. However, for other types of filled plots, such as graphing systems of inequalities, other techniques are far more flexible and correct. That will be covered in the electronic-only online appendix to this book "Plotting in Color, in 3D, and Animations," available on my webpage `www.gregorybard.com` for downloading.

```
P1 = text('$\int_0^2 (-1+\sqrt{x}) \, dx$', (0.5, 0.3),
                fontsize=20)
P2 = plot( -1+sqrt(x), 0, 2, fill=True)
P = P1 + P2
P.show()
```

As you can see from the screen-capture below, Sage knows which way to shade depending on if the function is positive or negative.

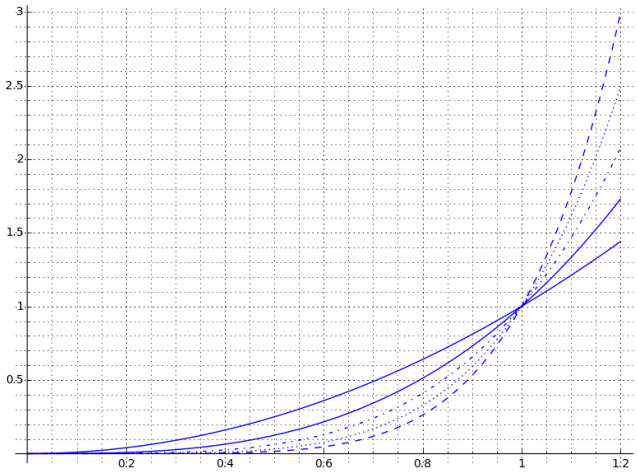


3.1.5. Dotted and Dashed Lines

Sometimes, when three or more functions are plotted on the same graph, it is nice to show which is which by some visual means. The better way of doing this is with color but another way is with dotted or dashed lines. Sage offers four variations, in addition to the normal solid line. The following bit of code

```
P1 = plot( x^2, 0, 1.2, gridlines='minor')
P2 = plot( x^3, 0, 1.2, linestyle = '-' )
P3 = plot( x^4, 0, 1.2, linestyle = '-.' )
P4 = plot( x^5, 0, 1.2, linestyle = ':' )
P5 = plot( x^6, 0, 1.2, linestyle = '--' )
P = P1 + P2 + P3 + P4 + P5
P.show()
```

will display each of those options simultaneously in the following graph:



On the other hand, plotting in color is covered in the electronic-only online appendix to this book "Plotting in Color, in 3D, and Animations," available on my webpage `www.gregorybard.com` for downloading.

3.2. Graphs of Some Hyperactive Functions

The function $\sin(1/x)$ wiggles like crazy near $x = 0$, but Sage knows this, and plots a lot of extra points. There are many other hyperactive functions in mathematics. Here are a few examples, and their Sage images. As you can see, Sage is imperfect but accomplishes the objective.

Plot 1: The function $f(x) = \sin(1/x)$ oscillates like crazy in the neighborhood $x = 0$. If you look carefully around $(0, 1)$ and $(0, -1)$, you'll see that the "blue smear" is not uniform. That has to do with how Sage plots functions. It will take the x-axis interval that was requested, and divide it up into 200 integrals. A random x-value is chosen from each interval. Under certain conditions, Sage can detect that the function is "complicated" and start using more points, closer together. However, it will always use a finite number of points, and so in "infinitely complicated" regions, there might non-uniformities. Furthermore, it means that sometimes a graph will look different when it is plotted twice in succession.

```
plot(sin(1/x), -0.5, 0.5)
```

Plot 2: The function $f(x) = exp(\frac{1}{2-x})$ has an interesting limit as x approaches 2. If approaching from the left (using x-values slightly less than two), the limit is infinity. If approaching from the right (using x-values slightly more than two), the limit is zero. Therefore, the limit when approaching 2 in general "does not exist." This is a nice example, because I find that students in *Calculus I* often believe that complicated limits are artificial inventions and are unlikely to ever come up.

```
plot(exp(1/(2-x)), -1, 4, ymin=-1, ymax=5)
```

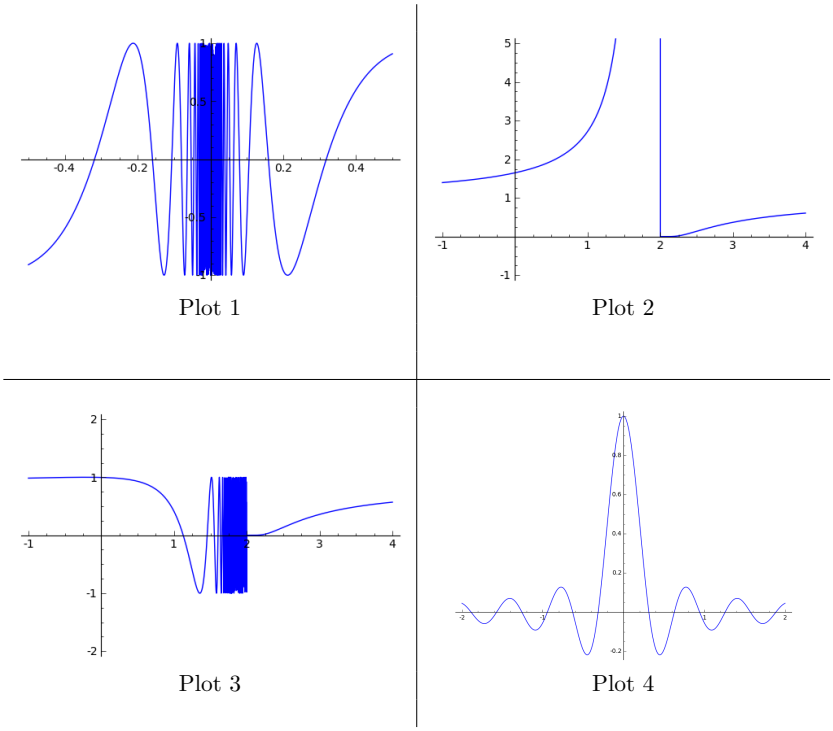
Plot 3: The function $f(x) = \sin(exp(\frac{1}{2-x}))$ is just the sine of the previous example. To the right of $x = 2$, this is like $\sin(0)$, and thus there is almost no oscillation. To the left of $x = 2$, this is like a sine wave where the frequency approaches infinity. It is interesting to look at this graph at various scales.

```
plot(sin(exp(1/(2-x))), -1, 4)
```

Plot 4: This function $f(x) = \frac{\sin(10x)}{10x}$ is interesting for several reasons. First, it is visually attractive. Second, it occurs in the study of wavelets, which is a subject in its own right. Third, many students will cancel the xs and work with $f(x) = \sin(10)/10$ instead, or perhaps cancel the 10s as well leaving only $f(x) = \sin$. Fourth, this function comes up in both mathematical analysis (as well as in signal processing) so frequently that it is given its own name:

$$f(x) = \text{sinc}(10x) = \frac{\sin(10x)}{10x}$$

```
plot(sin(10*x)/x, -2, 2)
```



3.3. Polar Plotting

The general idea behind polar coordinates is that instead of an x-coordinate and a y-coordinate, one has a radius r and an angle called θ. For any point, the r is the distance to the origin. The θ is defined as 0 being "east" or horizontal to the right, and positive angles represent counter-clockwise

motion. Thus, 90° or $\pi/2$ radians would be straight up or "north," and 180° or π radians would be straight left or "west." Similarly, negative angles represent clockwise motion. Thus, -90° or $-\pi/2$ radians would be straight down or "south."

Every direction of movement from the origin therefore has several names. For example,

$$\theta \in \left\{ \ldots, \frac{-22\pi}{4}, \frac{-15\pi}{4}, \frac{-7\pi}{4}, \frac{\pi}{4}, \frac{9\pi}{4}, \frac{17\pi}{4}, \frac{25\pi}{4}, \ldots \right\}$$

are values of θ that all represent "north east."

With that in mind, graphing a function in polar coordinates by hand is a bit challenging, because you do not know how many "laps" or "revolutions" around the origin are required before you look back on parts of the curve that are already drawn. Therefore, it is a bit non-trivial to figure out how to choose the values of θ that you want plotted.

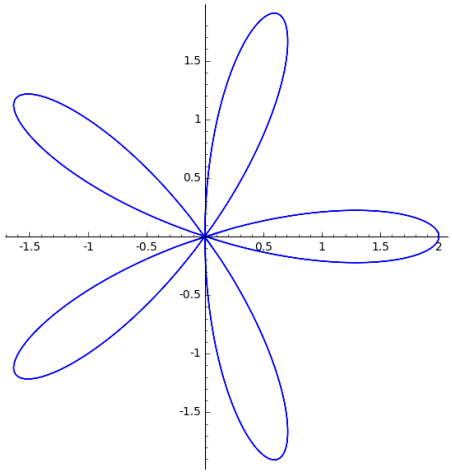
A function in polar coordinates takes a θ as input, and returns an r as output. Consider

$$r(\theta) = 2\cos 5\theta$$

which produces a graph called a 5-pedaled "rosette." To graph this in Sage, we would type

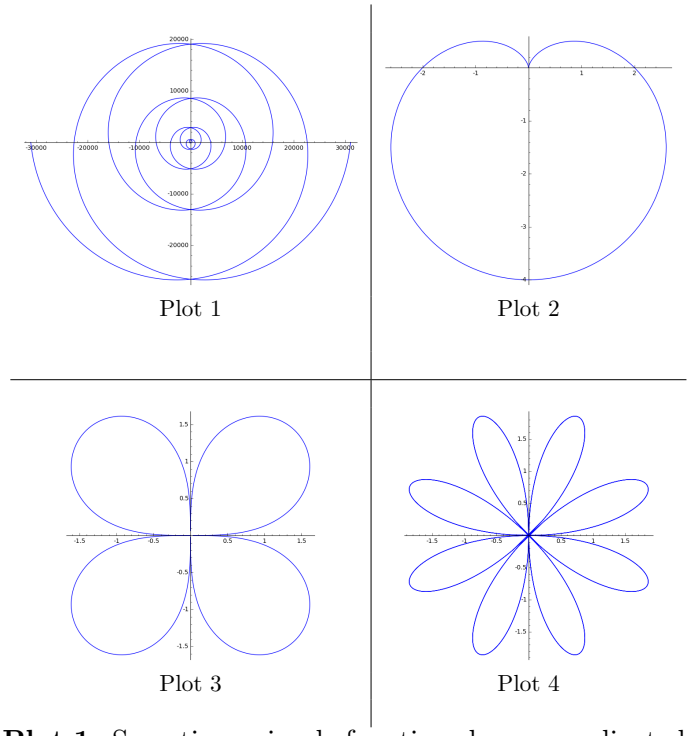
```
var("theta")
polar_plot( 2*cos(5*theta), (theta, 0, 2*pi) )
```

which results in the following image:



3.3.1. Examples of Polar Graphs

Here are four more interesting polar graphs.



Plot 1 Plot 2

Plot 3 Plot 4

Plot 1: Sometimes simple functions have complicated polar graphs. An example is $r(\theta) = \theta^3$ for $-10\pi < \theta < 10\pi$.

```
var("theta")
polar_plot( theta^3, (theta, -10*pi, 10*pi) )
```

Plot 2: This shape is called a "cardioid," and is given by

$$r(\theta) = 2 - 2\sin(\theta)$$

```
var("theta")
polar_plot(2 - 2*sin(theta), (theta, 0, 2*pi)  )
```

Plot 3: Here is a clover shape. Notice that the pedals have a very different shape from the rosettes. The function is $r(\theta) = 2\sqrt{|\sin(2\theta)|}$.

```
var("theta")
polar_plot( 2*sqrt(abs(sin(2*theta))), (theta, -pi, pi) )
```

Plot 4: This is an 8-pedaled rosette, given by $r(\theta) = 2\sin(4\theta)$.

```
var("theta")
polar_plot( 2*sin(4*theta), (theta, -2*pi, 2*pi) )
```

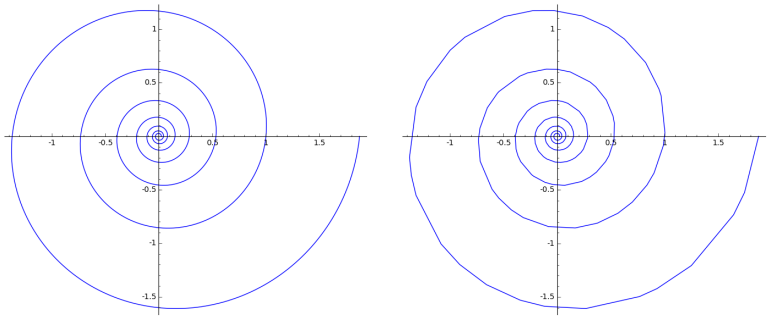
3.3.2. Problems that Can Occasionally Happen

The following curve produces a nice exponential spiral.

```
polar_plot( exp(theta/10), (theta,-12*pi,2*pi), plot_points=10000 )
```

This curve requires us to specify a large number points—in this case 10,000 points—for the plotting. To show why, consider the following two

images. For the image on the left, the `plot_points` option has been used, and on the right, it was removed—defaulting to only 200.



Plotted with 10,000 points Plotted with 200 points

As you can see, we get a much lower quality image when we plot only 200 points instead of 10,000 points. One way to understand this is to realize that the domain is $-12\pi < \theta < 2\pi$, and thus 200 points is only

$$200/14\pi \approx 4.54728\cdots \text{ points per radian}$$

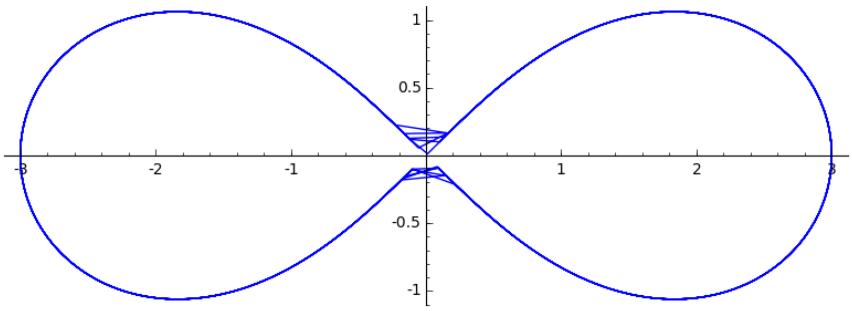
which is a problem because a radian is about 57 degrees. In contrast, if we use 10,000 points, we have $227.364\cdots$ dots per radian, which is 50 times as dense. If you look closely at the exponential spiral with 200 points, you'll see that Sage is drawing a piecewise linear curve—a sequence of line segments connecting the 200 estimated points.

Here is another example, with a very different reason for requiring a large number of points. This curve is called a Lemniscate. The code to produce the plot is

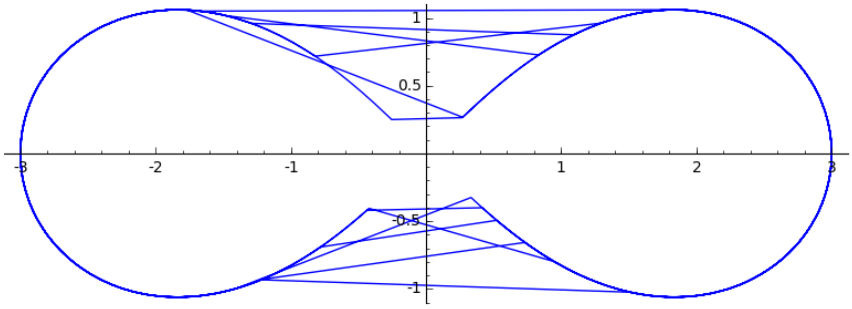
```
var("theta")
polar_plot( 3*sqrt(cos(2*theta)), (theta, -6*pi, 6*pi ),
          plot_points=10000)
```

In Figure 3.3.2, you can see the images produced with 10,000 points (above) and with 200 points (below). Partly, the phenomenon is exaggerated because we are using $-6\pi < \theta < 6\pi$ as the domain. We could have chosen $-\pi < \theta < \pi$ instead, and then we would have more points per radian.

You might be curious to know if there is some sort of condition whereby we could know, in advance, if this problem will occur or not. Actually, there is such a condition. The derivative $dr/d\theta$ will not exist at any points P when the curve is parallel with the line connecting P and the origin. Call such points "radially tangent points." At radially tangent points, the computer has to draw multiple pixels for one value of θ, and that only occurs for radially tangent points. That's why the algorithm doesn't quite work in those conditions. The analogy in rectangular coordinates would be when the tangent line is vertical. At such points dy/dx does not exist, and plotting often goes awry at those points—regardless if on a cheap graphing calculator



A Lemniscate Drawn with 10,000 Points.



A Lemniscate Drawn with 200 Points.

or in a sophisticated piece of software. We saw examples of that on Page 14 where we were graphing some rational functions.

3.4. Graphing an Implicit Function

Sometimes we write the equations for curves in a format such as

$$(x-1)^2 + (y-2)^2 = 9$$

instead of in a $y = f(x)$ format. The former is called an "implicit function" and the latter an "explicit function."

This is a good example, because we would need to write two functions

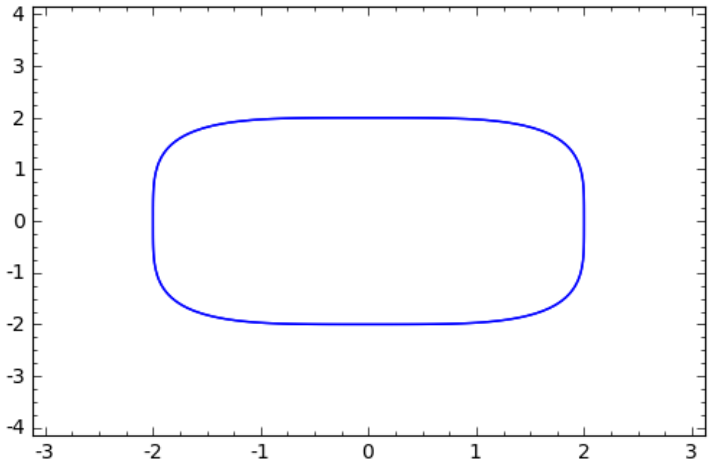
$$\begin{aligned} f(x) &= 2 + \sqrt{9-(x-1)^2} \\ g(x) &= 2 - \sqrt{9-(x-1)^2} \end{aligned}$$

to represent that curve. Therefore, Sage allows you to plot functions that are defined implicitly.

For example,

```
g(x,y)=x^4+y^4-16

implicit_plot(g, (x,-3,3), (y,-4,4) )
```

is a quartic curve favored by furniture designers for making conference room tables. As you can see, its mathematical formula is $x^4 + y^4 = 16$. Here is the plot:

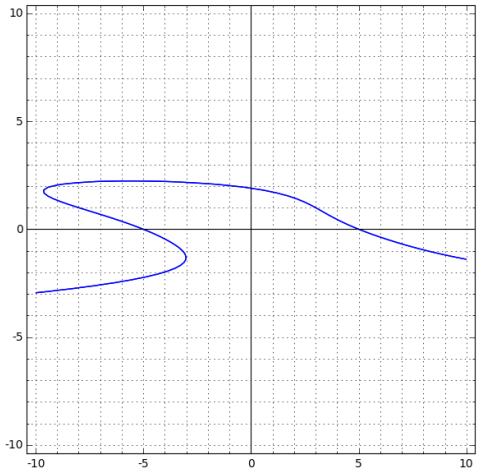


For other curves, such as

$$x^2 + y^5 + 5xy = 25$$

the explicit form would be even more tedious, if not impossible. The plot, below, is given by the commands

```
var("y")
implicit_plot( x^2 + y^5 + 5*x*y == 25, (x,-10,10), (y,-10,10),
          gridlines="minor", axes=True)
```



As you can see, this plot shows a nice background that looks like graph paper, using the `gridlines` option that we first saw in Section 1.4 on Page 8. The `axes=True` command is required to show the axes in an `implicit_plot`.

Note that implicit functions in three variables (x, y, and z) are discussed in the electronic-only online appendix to this book "Plotting in Color, in 3D, and Animations," available on my webpage `www.gregorybard.com` for downloading.

Backwards Compatibility:

The following syntax makes the `plot` command look like the `implicit_plot` command's syntax:

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

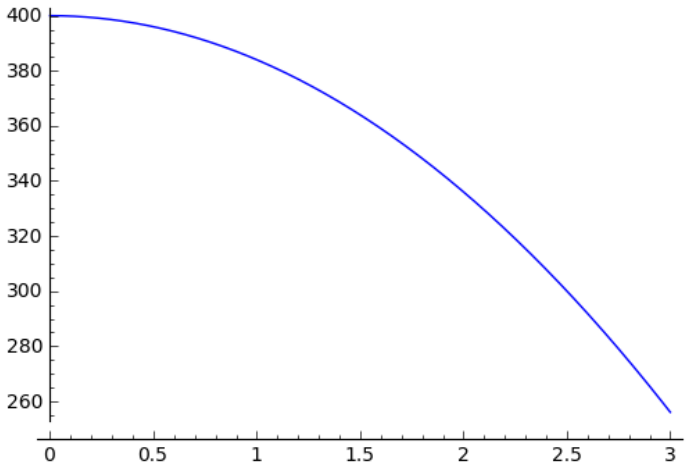
where as we would normally have typed

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

as in the original plotting section (Section 1.4 on Page 8). This useful for when you have to plot things in terms of t and not in terms of x. For example:

```
var('t')
plot( -16*t^2+400, (t, 0, 3) )
```

is the height of a bowling ball dropped off of a 400 foot building, t seconds after release. Here is the plot:



This same notation, of using `(t, 0, 3)` or `(x, -2, 2)` to mean $0 < t < 3$ and $-2 < x < 2$ will be used not only for `implicit_plot` but also for many other plot commands that we will now explore.

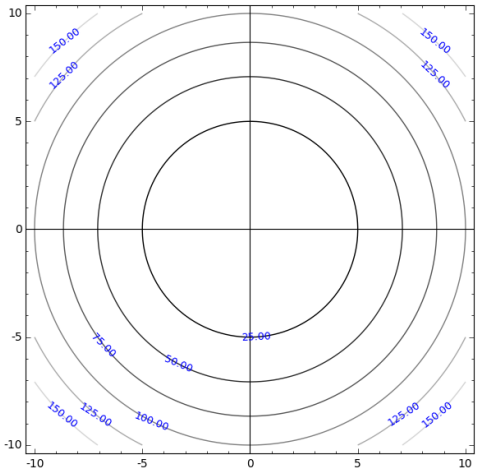
3.5. Contour Plots and Level Sets

One classic way of visualizing a function of two variables $f(x, y)$ is through a contour plot, which displays a collection of "level sets" for that function.

A level set is the set of points (x, y) such that $f(x, y) = z$ for some fixed z. For example, if

$$f(x, y) = x^2 + y^2$$

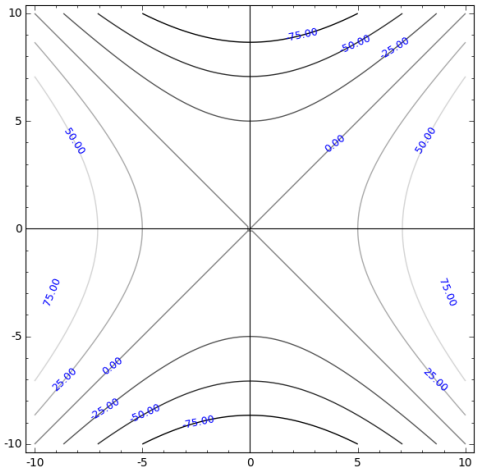
then the following plot shows the level sets for $z \in \{25, 50, 75, 100, 125, 150\}$.



The code for making that image is

```
var("y")
contour_plot( x^2 + y^2, (x, -10, 10), (y, -10, 10),
          fill=False, axes=True, labels=True )
```

Here is another contour plot, for $g(x, y) = x^2 - y^2$. To generate this image, all I did was change the "+" into a "−" between x^2 and y^2 in the previous code.

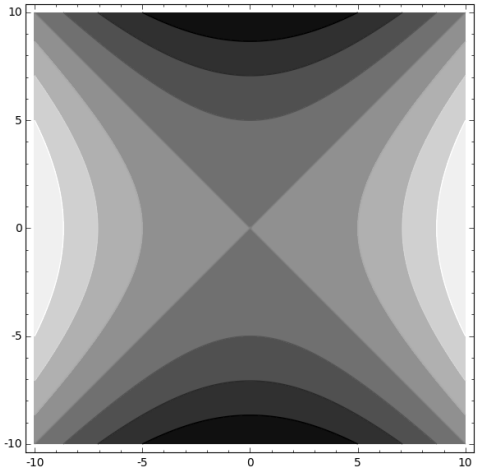


These are the old-fashioned contour plots that would have been found in textbooks during the 20th century. Modern contour plots use shading, which usually makes them easier to read, but at an increased cost of ink. The following code

```
var('y')
```

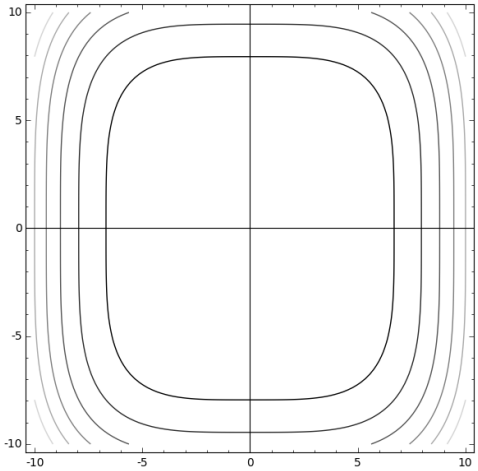
```
contour_plot( x^2-y^2, (x, -10, 10), (y, -10, 10), plot_points=200)
```

generates an image for the same function as the previous image, $g(x, y) = x^2 - y^2$, but in the shaded style.



In these shaded contour plots, the darker regions represent f values that are smaller, and the lighter regions indicate f values that are larger. By the way, the `plot_points` option helps Sage figure out how carefully drawn you want the contour plot to be. I usually use 200 or 300. The default is 100, which is too small in my opinion.

In all fairness, I must note that sometimes the shading does more harm than good. The following contour diagram is perfectly readable without shading.



The code that generated the previous contour plot was

```
var("y")
contour_plot( x^4 + (y)^4/2, (x, -10, 10), (y, -10, 10),
     fill=False, axes=True )
```

On the other hand, if one backspaces over the "`fill=False`" and "`axes = True`" commands, the resulting shaded contour plot has a giant black region occupying the large central portion of the plot, inside the innermost oval. This renders the graph almost unreadable—try it if you like.

In order to explore the relationships between some three-dimensional objects and their contour diagrams, I have made an interactive web page (sometimes called an interact or an applet) to show some complex shapes both in 3D plots and in contour plots, simultaneously.

3.5.1. An Application to Thermodynamics

The following example is taken from *Elementary Differential Equations and Boundary Value Problems*, by William Boyce and Richard DiPrima, 9th edition, published by Wiley in 2009. It is Example 1 of Section 10.5, "Separation of Variables: Heat Conduction in a Rod." This extended example will require some techniques from Section 4.1 and Section 4.19. I sincerely hope it is intelligible without having studied those sections too closely, but if the reader finds this example confusing, it might be profitable to read those sections. Before we begin it is necessary that I apologize for the fact that the typesetting here did not permit me to attach units to constants in all cases, as is the practice among physicists.

The function $u(x,t)$ below tells us the temperature u at time t and position x inside of a rod. This rod is 50 cm long, insulated on the outside, and is initially uniformly at the temperature 20°C or 68°F. The ends are abruptly exposed to a temperature of 0°C or 32°F at $t = 0$ and held there for $t > 0$. As you can see, the function is defined by a series

$$u(x,t) = \frac{80}{\pi} \sum_{n=1,3,5,\ldots}^{n=\infty} \frac{1}{n} e^{-n^2\pi^2\alpha^2 t/2500} \sin\frac{n\pi x}{50}$$

but the series variable n runs over all odd positive integers.

Instead, let's define $n = 2j+1$. Then as j runs through 0, 1, 2, 3, ..., n will run through 1, 3, 5, 7, ..., producing all odd positive integers. This will allow us to use Sage's `sum` command. Furthermore, we'll just let j run from 0 to 100. After all, when $j = 101$, we will have $n = 203$, and that term will have $e^{-203^2\pi^2}$ in it, which is going to make that term completely negligible. In reality, we could stop at $j = 10$ or $j = 20$ and probably produce the same graph, but computers do not mind doing a lot of tedious work, so we'll compromise at $j = 100$. Boyce and DiPrima use $\alpha = 1$, and so we will do the same. Now we have

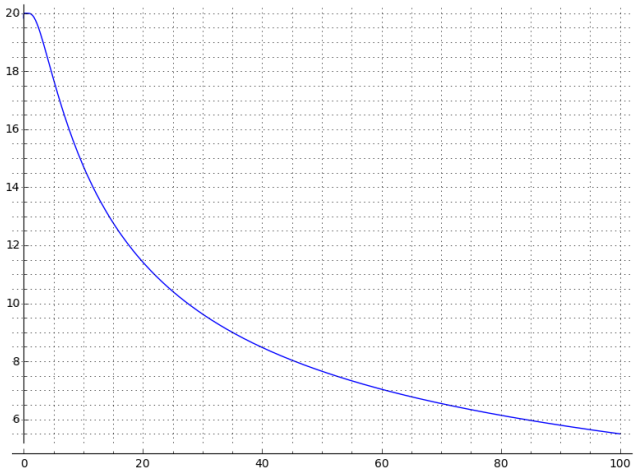
$$u(x,t) = \frac{80}{\pi} \sum_{j=1}^{n=\infty} \frac{1}{2j+1} e^{-(2j+1)^2\pi^2 t/2500} \sin\frac{(2j+1)\pi x}{50}$$

Since this is a function of two variables, one way we could visualize this is by "half-evaluation." We could plot $u(5,t)$ for $0 < t < 100$ and get an

idea of what the temperature is like over time, 5 cm from one end of the rod. To do this, we would type

```
var("j t")

u(x, t) = (80/pi)*sum( (1/(2*j+1))*exp(-(2*j+1)^2*pi^2*t/2500.0)
          * sin((2*j+1)*pi*x/50), j, 0, 100)

plot( u(5, t), t, 0, 100, gridlines="minor" )
```

This produces the following plot:

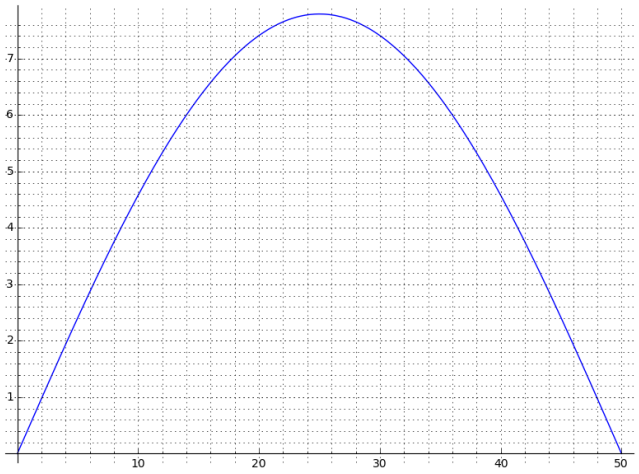


A few quick comments are in order. We used `2500.0` instead of `2500` to force Sage to evaluate the expression numerically, and not exactly—otherwise the resulting exact, algebraic expression is far too complex, and the computer takes ages to perform the computation. Second, the `u(5, t)` is an example of half-evaluation (see Page 127). Third, the syntax for summing a series is given on Page 182.

Alternatively, we could plot $u(x, 300)$ and get an idea of what the temperature is like throughout the rod, at 300 seconds or 5 minutes after the cooling has begun. We would replace the last line of the above code with

```
plot( u(x, 300), x, 0, 50, gridlines="minor" )
```

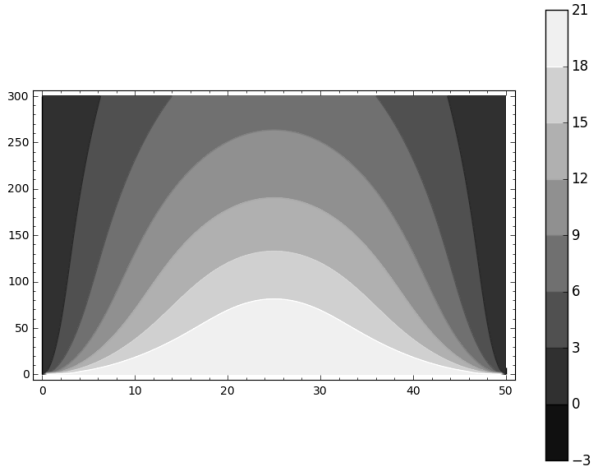
That produces the image:



However, a contour plot is much more powerful than any of these. If you have a contour plot, and draw a vertical line, looking only along that vertical line, you find out how temperature varies over time at a specific point. If you draw a horizontal line, looking only along that horizontal line, you will find out how temperature varies over the rod at a specific time. To make the contour plot, we replace the last line of the above code with

```
contour_plot( u(x,t), (x,0,50), (t,0,300), aspect_ratio=0.1,
          colorbar=True )
```

That code gives us the following contour plot:



As you can see, a "color bar" or grayscale legend has been attached to tell us what the various shades of gray indicate, in terms of degrees Celsius. That is quite useful here in thermodynamics, but it would not necessarily have been relevant in the pure mathematics examples that we saw earlier in this section.

Last but not least, the `aspect_ratio` option needs to be explained. When plotting geometric objects, such as the two-variable functions that

we saw moments ago, it is important that the same scale be used on the x-axis and the y-axis. Otherwise the angles will be off and other distortions will occur. If we were to remove the `aspect_ratio=0.1` option, Sage would observe that we are plotting 300 in one variable, and 50 in another variable. Therefore, the graph would be six times narrower than it is tall, to maintain the same scale. The best way to see this is to backspace over the `aspect_ratio=0.1` option, and hit evaluate. Instead, with the option, the 50 on the x-axis is interpreted as 500, and we obtain a graph that is 1.6 times wider than it is tall. That is a graph much easier to read. We should not feel bad about making this change, because 50 cm and 300 s are in totally different units, and thus the notion of "the same scale" is meaningless.

3.5.2. Application to Microeconomics (Cobb-Douglas Equations)

The following is an example of a Cobb-Douglas production function

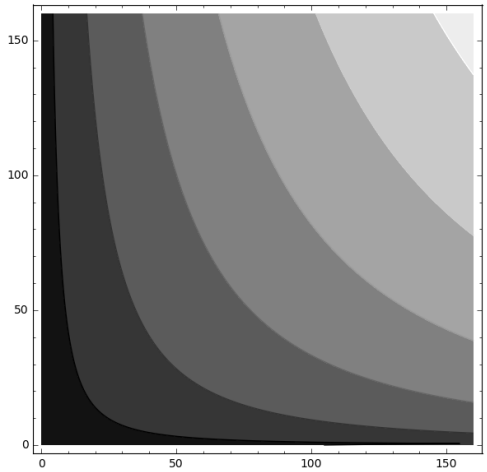
$$f(K, L) = 1400K^{0.51}L^{0.32}$$

which was derived from hypothetical data for a small business that produces concrete pipes. The model was found in *Linear and Non-Linear Programming with Maple: An Interactive, Applications-Based Approach* by Paul Fishback, published by CRC Press in 2010. The function appears in several places throughout the book, but is derived from data as the primary example of Section 7.3.5, "Non-Linear Regression."

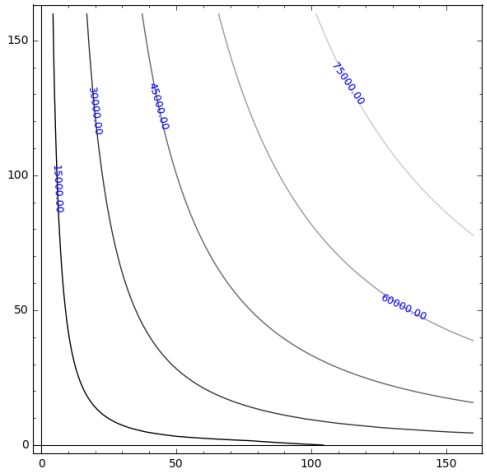
Using the code:

```
var("y")
contour_plot( 1400*x^0.51*y^0.32, (x, 0, 160), (y, 0, 160),
          plot_points=300)
```

we get the following shaded contour plot, which is moderately informative.



However, the much more readable unshaded plot



was produced by the following code:

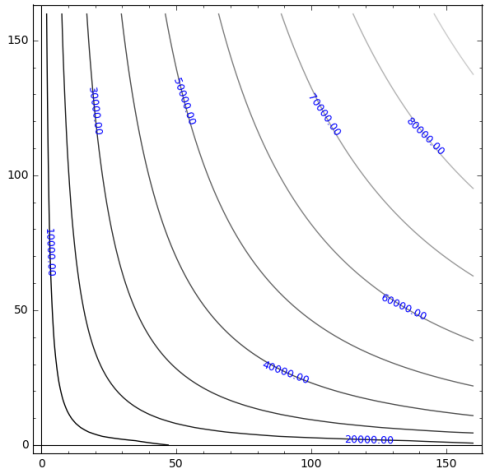
```
var("y")
contour_plot( 1400*x^0.51*y^0.32, (x, 0, 160), (y, 0, 160),
          fill=False, axes=True, labels=True )
```

Forcing Particular Values

Sometimes it is useful to force Sage to choose particular contour lines. If certain values are very important, or called for by a homework exercise, then one might want to dictate to Sage what specific z-values are used for the level sets. Here is an example.

```
var("y")
contour_plot( 1400*x^0.51*y^0.32, (x, 0, 160), (y, 0, 160),
          fill=False, axes=True, labels=True,
          contours=[10000, 20000, 30000, 40000, 50000, 60000,
          70000, 80000, 90000, 100000] )
```

produces the following image:



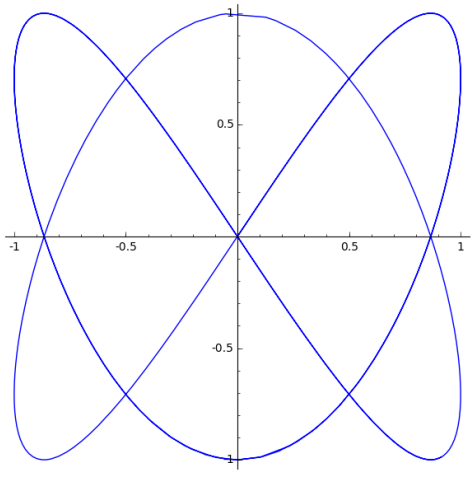
3.6. Parametric 2D Plotting

Mathematicians use the term "parametric function" to refer to situations where the x-coordinate of a point is given as $f(t)$, whereas the y-coordinate of a point is given as $g(t)$. Therefore, for any point in time t, you know the x-coordinate and the y-coordinate. This is very useful for describing more advanced types of motion in physics, including celestial mechanics—the movement of planets, asteroids and comets.

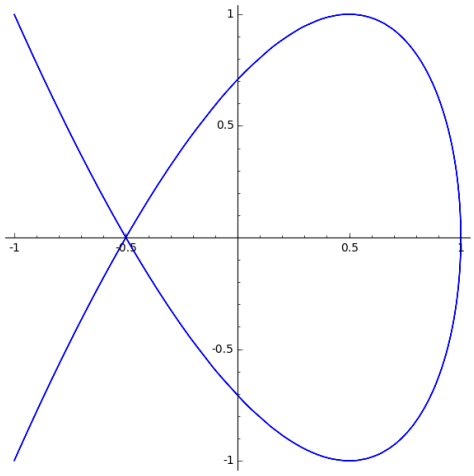
A really neat example is the Lissajous Curve, named for Jules Antoine Lissajous (1822–1880), but actually discovered by Nathaniel Bowditch (1773–1838). Since Lissajous made a detailed study of them, the curves were named for him. These curves are of the form $x = \sin(at + b)$ and $y = \sin(ct + d)$, with a and c usually restricted to the integers, though some textbooks allow a multiplicative constant in front of each sine function, or rational values for a and c. In any case, if you have $a = c$ then it is just an ellipse or circle. However, $a \neq c$ makes for much more interesting figures. Here is some code for the $a = 2$ and $c = 3$ case.

```
var("y t")

f(t) = sin(2*t)
g(t) = sin(3*t)

parametric_plot( ( f(t), g(t) ), (t,0,2*pi) )
```

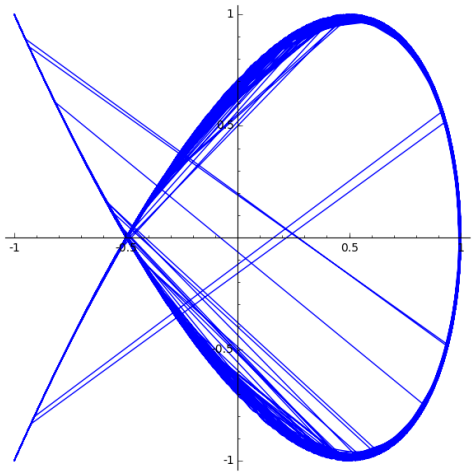
That code produces the following image:



Lissajous curves are very important in tuning oscilloscopes as well as tuning audio systems. If we change $f(t)$ to be instead $\cos(2t)$, we get a very different figure.



The phenomenon that we saw with polar coordinates on Page 101, where we had too few points per radian, can also occur in parametric equations. Consider changing the domain to $-100 < t < 100$ in the previous example, taking care to keep $y = \cos(2t)$. To be explicit, that means change `(t, 0, 2*pi)` into `(t, -100, 100)`. The resulting output is extremely poor, because the points are far apart. Since the domain is length 200 in units of t, and the default number of points plotted is 200, each point used to construct the plot is 1 unit of t apart. When we had the interval being of length 2π, the points are $\pi/100 \approx 0.0314159$ units of t apart, rather a large difference.



It is an excellent exercise for students to try to convert the parametric form of a function into an implicit function—if possible. However, it is not possible (using the elementary functions) for some curves. Also, the calculus of parametric curves is straightforward but significantly different than that of explicit functions. Most calculus textbooks cover the parametric functions topic but some instructors skip it—which can be a problem for engineering or physics students who encounter parametric functions or equations in higher

level classes. If the reader would like to learn more, I recommend Ch. 9.2 and Ch. 9.3 in Soo Tan's *Calculus with Early Transcendentals*, 1st Edition, published by Cengage in 2011.

3.7. Vector Field Plots

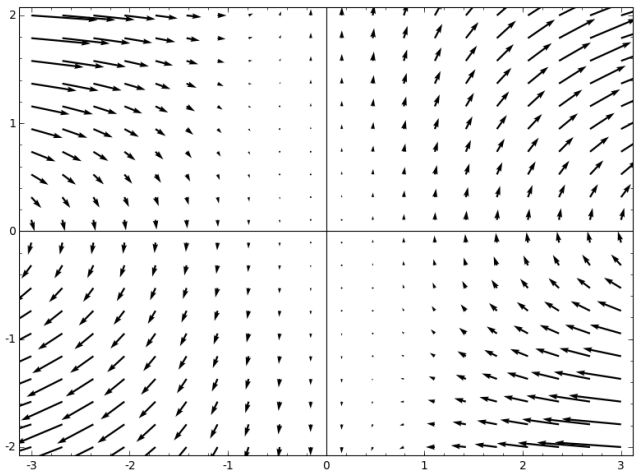
First, it might be a good idea for us to remind ourselves what a vector field plot looks like. Here's an example, where we're plotting the following vector-valued function:

$$\vec{f}(x,y) = \begin{bmatrix} \sin x \\ \cos y \end{bmatrix}$$

To make the vector field plot, we'd type

```
var("y")
plot_vector_field( (sin(x), cos(y)), (x, -10, 10), (y, -10, 10),
          plot_points=30 )
```

Note, that's another example of how Sage knows that x is a variable, but we have to remind it that all other letters are variables, declaring them with the `var` command. Here we're saying that the x-value of our vector-valued function is `sin(x)`, and the y-value of our vector-valued function is `cos(y)`. Then the `(x, -10, 10)` notation means that the x-range we want is $-10 < x < 10$, and naturally the `(y, -10, 10)` indicates $-10 < y < 10$. The `plot_points` parameter tells Sage how many vectors to draw in each row and in each column of the plot. I'm sure you'd agree that making such a plot, with 900 separate arrows, would be exceptionally tedious to do by hand. Now we have the following plot



Just as $\vec{f}(x,y)$ returns a vector for each point (x,y), we have (at a bunch of points in the coordinate plane) an arrow, representing some vector. The direction is very faithfully reproduced, and the length of the arrow is a rough representation of the magnitude of that vector.

How can we interpret such plots? One way is that if a small object were placed in the coordinate plane, at each instant of time, it would move in the direction pointed to by the arrow under the object. The only assumptions are that the mass and radius of the object be tiny. Another interpretation involves gradients, which we will now explore.

3.7.1. Gradients and Vector Field Plots

One reason to make a vector field plot is to study the gradient of a function. We can see that if

$$g(x, y) = (\sin y) - (\cos x)$$

then the partial derivatives would become

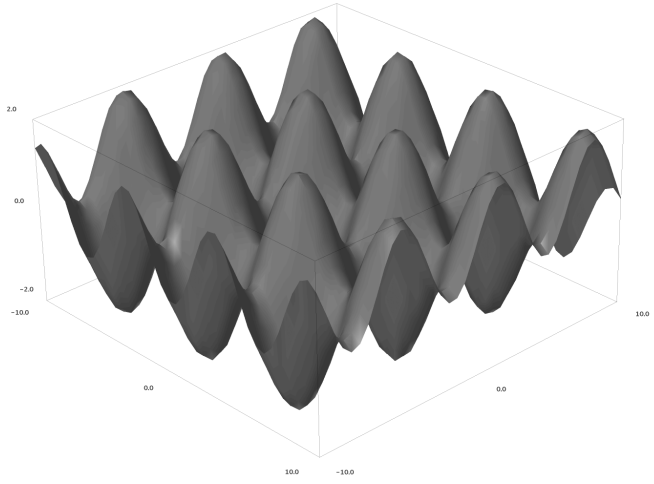
$$\frac{\partial g}{\partial x} = \sin x \qquad \text{and} \qquad \frac{\partial g}{\partial y} = \cos y$$

which would make the gradient

$$\nabla g = \begin{bmatrix} \frac{\partial g}{\partial x} \\ \\ \frac{\partial g}{\partial y} \end{bmatrix} = \begin{bmatrix} \sin x \\ \cos y \end{bmatrix} = \vec{f}(x, y)$$

As you can see, that's the same vector-valued function which we just plotted. If we wanted to visualize $g(x, y)$ as the z-coordinate of (x, y) we could cause a 3D-plot using

```
g(x,y) = sin(y) - cos(x)

plot3d( g, (x,-10,10), (y,-10,10) )
```



As you can see, we get a patterned object not unlike an egg carton. Now look at the vector field plot again. There are some spots where a bunch of arrows are all pointing to the same point. If you put a marble in the neighborhood of that point, it would be drawn toward that point, and stay there. Those are points where ∇g is the zero-vector, and corresponds

to a local minimum of the egg carton. There are also spots where a bunch of arrows are all pointing away from a particular point. If you put a marble in the neighborhood of that point, it would be repelled away from that point and not return. Those are points where ∇g is also the zero-vector, but they correspond to a local maximum of the egg carton. An applied mathematician or physicist would say that these are attractive and repulsive equilibrium points, for the local minimum and local maximum cases, respectively.

How can you distinguish between those local minima and maxima? Well, you could have Sage generate a vector field plot and make the identification visually. Or, you could consider the Hessian matrix (in this case a 2×2 matrix of second partial derivatives), and then compute the determinant of that matrix. Clearly, one task is considerably more difficult than the other.

3.7.2. An Application from Physics

Imagine three electric charges in the coordinate plane. Charge 1 is glued to the origin, and Charge 2 is glued to the point $(2, 0)$. The third charge is mobile, at the point (x, y). We might like to know what the total force of electric charge is on the mobile third charge. To make things more interesting, let's imagine that charges are +2, +1, and -1 for Charge 1, Charge 2, and the mobile charge. The unequal charges will make the problem more interesting. A fantastic way to visualize this is with a vector field plot.

Before we continue, let's briefly remind ourselves of Coulomb's Law. The force $\vec{F}$ (in Newtons) of a charge of q_0 Coulombs upon a charge of q_1 Coulombs is given by

$$\vec{F} = -\left(\frac{1}{4\pi\epsilon_0}\right)\left(\frac{q_1 q_0}{||\vec{r}||^3}\right)\vec{r}$$

where $\vec{r}$ is the vector from q_1 to q_0, where ϵ_0 is constant (called "the permittivity of free space"), and where π has its usual meaning. Some textbooks define $\vec{r}$ to be the vector from q_0 to q_1, in which case we would drop the minus sign at the front of the right-hand side. (By the way, the notation $||\vec{r}||$ means the magnitude, or length, of the vector $\vec{r}$.)

Some readers might be surprised to see the cube in the denominator. That's because we are working with vectors. If we wanted only to work with scalars, i.e. the real number expressing the magnitude of the force, we'd have instead

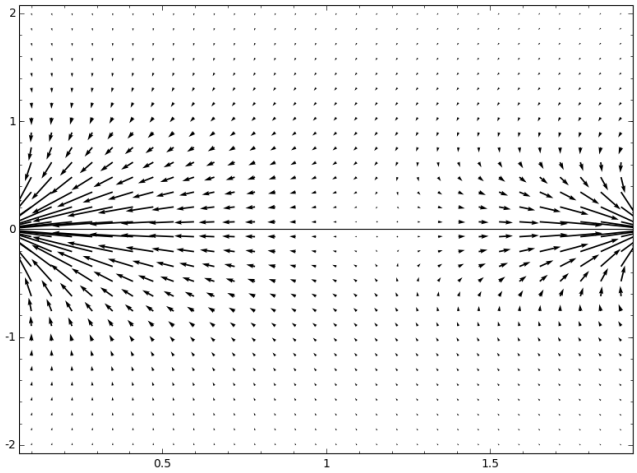
$$||\vec{F}|| = \left|-\frac{1}{4\pi\epsilon_0}\right|\left(\frac{|q_1||q_0|}{||\vec{r}||^3}\right)||\vec{r}|| = \frac{1}{4\pi\epsilon_0}\left(\frac{|q_1||q_0|}{||\vec{r}||^2}\right)$$

giving us the familiar inverse-square law.

We're now ready to analyze the code (found in Figure 1) which was used to generate the following plot.

```
plot_vector_field( net_f, (x, -1, 4), (y, -2, 2), plot_points=30 )
var("y")

x1= 0
y1= 0
x2= 2
y2= 0

r_vec_1 = vector( [ x1-x, y1-y ] )
r_vec_2 = vector( [ x2-x, y2-y ] )

q0= -1
q1= 2
q2= 1

f1 = - q0*q1*r_vec_1  /  r_vec_1.norm()^3
f2 = - q0*q2*r_vec_2  /  r_vec_2.norm()^3

net_f = f1 + f2

plot_vector_field(net_f, (x, 0.1, 1.9), (y, -2, 2), plot_points=30)
```

FIGURE 1. The Code for the Physics Application of Vector Field Plots



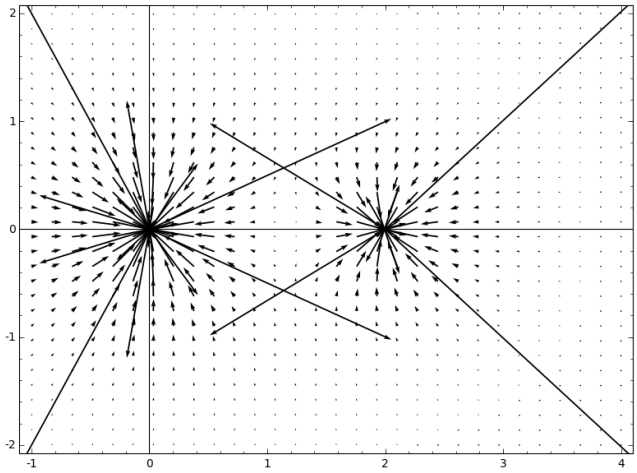
First, we define `x1` and `y1` to be the coordinates of Charge 1, at the origin. Second, we define `x2` and `y2` to be the coordinates of Charge 2, located at the point $(2, 0)$. It is useful for us to have these values isolated at the top of the program (rather than substituted into the middle of formulas, later in the program), because it will enable us to experiment by adjusting the numbers slightly.

Third, we define two vectors. The vector `r_vec_1` goes from (x, y) to (x_1, y_1). Likewise, the vector `r_vec_2` goes from (x, y) to (x_2, y_2). Whenever you have to define a vector, whether in Sage or in any other mathematical situation, it is convenient to remember the trick "head minus tail." If you visualize `r_vec_1` as an arrow, then the arrowhead would be at (x_1, y_1), and the tail feathers would be at (x, y).

Fourth, we define the charges of the mobile charge, Charge 1 and Charge 2 as `q0`, `q1`, and `q2`. Fifth, we apply Coulombs law. For the force between Charge 1 and the mobile charge, we compute `f1` and likewise `f2`. As you can see, I dropped all the constants, because they won't affect[1] the directions of the vectors. Sixth, `net_f` is the sum of those two forces, giving the total force on the mobile charge. Note that we had no need of computing the force of Charge 1 upon Charge 2, nor the equal and opposite force of Charge 2 upon Charge 1, because they aren't relevant to the mobile charge. Last but not least, we use the `plot_vector_field` command, to plot the vector field for $0.1 \leq x \leq 1.9$ and $-2 \leq y \leq 2$.

Like Icarus, Don't get Too Close!

While the charges are located at $(0, 0)$ and $(2, 0)$, you'll observe that we've deliberately set the x-values to exclude $x = 0$, $x = 2$, and their immediate neighborhood. If you change `(x, 0.1, 1.9)` to `(x, -1, 4)` in the last line of the code of Figure 1, you'll see that there are some crazy and huge vectors there that appear to be signaling nonsensical conclusions. In particular, it looks like some of those large vectors are pointing the wrong way.



What's actually happening here is that when (x, y) gets too close to one of the charges, the distance gets very small. That means that the denominator, which is cubed, gets extremely tiny. That makes the force absolutely huge. The vector then becomes so long that it ends up taking up

[1]It was also for this reason that we never bothered with units in this problem—the units will not affect the directions of the vectors.

far more space in the plot than it should, and creates this visually misleading result. In a Sage vector field plot, the tail of the arrow represents the location which the arrow is describing. However, the eye seems to imagine that the arrowhead represents that location. Normally, this won't come up often.

Mathematically speaking, $(0, 0)$ and $(2, 0)$ are the poles of the total force function, because the function divides by zero there. In general, you will get a much better vector field plot if you exclude the poles from the region being graphed.

A Challenge:

You are now ready to attempt the mini-project on Electric Field Vector Plots, which is given on Page 90.

3.7.3. Gradients versus Contour Plot

The following example very nicely shows the relationship between contour plots and the vector field plot of the gradient. This example was created by Augustine O'Keefe. We are considering the function

$$f(x, y) = \cos(x) - 2\sin(y)$$

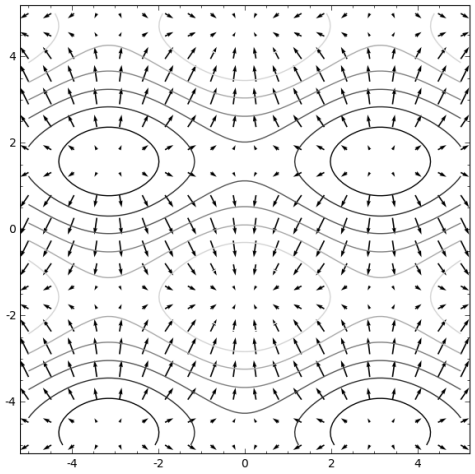
and its gradient, which is

$$\nabla f(x, y) =< \frac{\partial f}{\partial x},\ \frac{\partial f}{\partial y} >=< -\sin(x),\ -2\cos(y) >$$

We will draw the contour plot of $f(x, y)$, as well as the vector field plot of the gradient, $\nabla f(x, y)$, but superimposed on the same image. The code that we use is as follows:

```
f(x,y) = cos(x)-2*sin(y)
gradient = derivative( f )

P1 = plot_vector_field( gradient, (x,-5,5), (y,-5,5) )

P2 = contour_plot(f, (x,-5,5), (y,-5,5), fill=False)

show( P1 + P2 )
```

That code produces this image:



Here we can see what is like a collection of hills and valleys (the contour plot), almost in the style of geological contour maps used by geologists and boy scouts. The arrows, representing the gradient, represent the direction of movement that a drop of rain would take, moving from higher altitude to lower altitude. We can clearly see where the flooding would be in a major rainstorm. Last but not least, we can definitely see that the gradient vector is always perpendicular to the level-set curves (the curves that comprise the contour plot.)

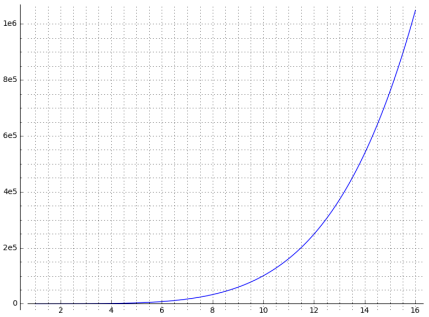
3.8. Log-Log Plots

In the middle of the twentieth century, it was not uncommon to find "log-log paper" available for purchase at university bookstores. This is graph paper where the gridlines are spaced in a peculiar way: each horizontal and vertical line is a logarithmic scale. For example, instead of some particular inch representing $x = 1$ to $x = 2$ divided into steps of perhaps 1/10th (e.g. 1, 1.1, 1.2, 1.3, ..., 2), a particular inch might represent $x = 10$ to $x = 100$. The subdivisions would represent 10^1, $10^{1.1}$, $10^{1.2}$, $10^{1.3}$, ..., 10^2. The next inch would represent $x = 100$ to $x = 1000$ and the previous inch would represent $x = 1$ to $x = 10$.

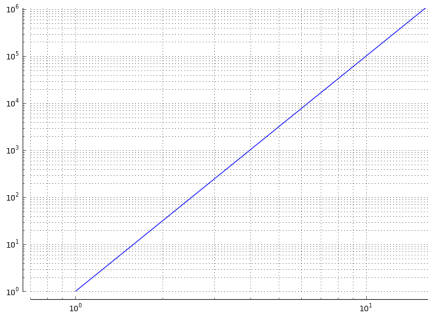
The first question would be "why was this done?" The purpose of a log-log plot is to enable the computations of power laws by means of regression, or perhaps to simply display a power law. For example, the volume of a sphere is proportional to the cube of its radius; the stopping distance of a car is proportional to the square of its speed; the energy of the return of a radar signal is proportional to the inverse fourth power of the distance to the target; many other examples exist. While c and n can vary, all these relationships are of the form $y = cx^n$.

In the era of graphing calculators and computer algebra packages, log-log paper is extremely hard to find. However, Sage can make plots in a

log-log fashion. First, let's illustrate the effect by considering $y = x^5$, which is clearly a power law with $c = 1$ and $n = 5$.



Ordinary Coordinates



Log-log Coordinates

The plot on the left was given by

```
plot( x^5, (x, 1, 16), gridlines="minor"  )
```

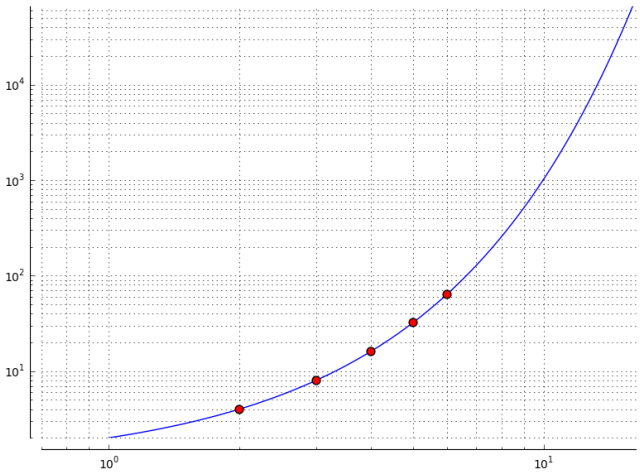
and that of the right was given by

```
plot_loglog( x^5, (x, 1, 16), gridlines="minor"  )
```

The following code plots a function $y = 2^x$ along with five data points, namely $\{(2, 4),\ (3, 8),\ (4, 16),\ (5, 32),\ (6, 64)\}$.

```
plot_loglog( 2^x, ( x, 1, 16 ), gridlines="minor" ) +
    scatter_plot( [ (2,4), (3,8), (4,16), (5,32), (6, 64) ],
    facecolor='red' )
```

Below is the plot generated by the above code.



For more information about `scatter_plot` see Section 4.9 on Page 157. Since this transformation of ordinary two-dimensional space (namely $\mathbb{R}^2$) into the log-log plane was so useful, you might be curious if there is an analogy for three dimensional space (namely $\mathbb{R}^3$). There is, and it is called log-log-log space. This comes up in the derivation of Cobb-Douglas style formulas in macroeconomics. In general, those equations are of the form

$$P = cK^aL^b$$

where P is production, K is capital invested, and L is labor invested. The coefficients a, b, and c are found by fitting a best-fit plane (the three-dimensional analog of a best-fit line) but in log-log-log space instead of in ordinary three-dimensional space. See Section 3.5.2 on Page 110 for an example of a Cobb-Douglas function derived from data.

3.9. Rare Situations

Here we have two rare situations that might come up, or might not. One is a necessary flaw in the way Sage handles the odd roots of negative numbers. The other has to do with functions that have x-values missing from their domains.

Background on Cube Roots

When we say $\sqrt[3]{64} = 4$, what do we really mean? Of course, this is just a way of writing the fact that $4^3 = 64$. However, when the complex numbers enter the picture, things get a bit more complicated.

If I ask for the cube root of 8, there are three numbers z in the complex plane such that $z^3 = 8$. Specifically, they are

$$\begin{aligned} z &= 2\cos(0^\circ) + i2\sin(0^\circ) = 2 + 0i = 2 \\ z &= 2\cos(120^\circ) + i2\sin(120^\circ) = -1 + i\sqrt{3} \\ z &= 2\cos(240^\circ) + i2\sin(240^\circ) = -1 - i\sqrt{3} \end{aligned}$$

Let's take a brief moment to verify that last one, by cubing it carefully.

$$\begin{aligned} \left(-1 - i\sqrt{3}\right)^3 &= \left(-1 - i\sqrt{3}\right)\left(-1 - i\sqrt{3}\right)^2 \\ &= \left(-1 - i\sqrt{3}\right)\left(1 + 2i\sqrt{3} + i^2 3\right) \\ &= \left(-1 - i\sqrt{3}\right)\left(1 + 2i\sqrt{3} - 3\right) \\ &= \left(-1 - i\sqrt{3}\right)\left(-2 + 2i\sqrt{3}\right) \\ &= (-1)(-2) + (-1)(2i\sqrt{3}) + (-i\sqrt{3})(-2) + (-i\sqrt{3})(2i\sqrt{3}) \\ &= 2 - 2i\sqrt{3} + 2i\sqrt{3} - 2i^2(3) \\ &= 2 - (2)(-1)(3) \\ &= 2 + 6 = 8 \end{aligned}$$

Therefore, the set of complex numbers that cube to 8 would be

$$\left\{2,\ -1 + i\sqrt{3},\ -1 - i\sqrt{3}\right\}$$

However, of these three, the first is real—normally that's "the one we want" (at least, usually.) Similarly, if I ask for the cube root of -8, there are three numbers z in the complex plane such that $z^3 = -8$. Specifically, they are

$$\begin{aligned} z &= -2\cos(0^\circ) - i2\sin(0^\circ) = -2 - 0i = -2 \\ z &= -2\cos(120^\circ) - i2\sin(120^\circ) = 1 - i\sqrt{3} \\ z &= -2\cos(240^\circ) - i2\sin(240^\circ) = 1 + i\sqrt{3} \end{aligned}$$

Therefore, the set of complex numbers that cube to -8 would be

$$\left\{-2,\ 1 - i\sqrt{3},\ 1 + i\sqrt{3}\right\}$$

and as you can see, it is just the previous set times -1. Again, of these three solutions, the first is real, and normally that's "the one we want" (at least, usually.) Now it is noteworthy that I can get all three solutions with the following command

```
solve( x^3 == 8, x )
```

and of course, it works fine with `== -8` also.

The issue is that for certain extremely important topics, including algebraic number fields and finite fields (branches of abstract algebra), it is important that Sage return one of the two complex roots when you type

```
print -8^(1/3)
```

even though we usually desire the real root.

That means if you type

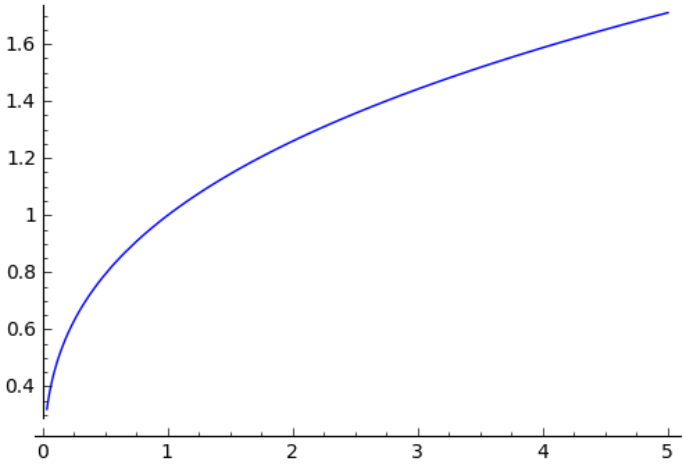
```
plot( x^(1/3), -5, 5 )
```

then you'll only get a graph showing $0 < x < 2$, along with the error message

```
verbose 0 (2414: plot.py, generate_plot_points)
WARNING: When plotting, failed to evaluate function at 100 points.
verbose 0 (2414: plot.py, generate_plot_points)
Last error message: 'negative number cannot be raised to a fractional power'
```

because Sage is returning the complex roots (which are not visible on the real plane) for the $x < 0$ values. As you can see, Sage is explicitly reprimanding you for raising a negative number to a fractional power.

The graph is given below:



Now I'll tell you how to circumvent this issue.

Plotting Odd Roots of Negative Numbers

To plot the negative real cube root, you have to use the following:

```
plot(lambda x : RR(x).nth_root(3), (x,-1, 1))
```

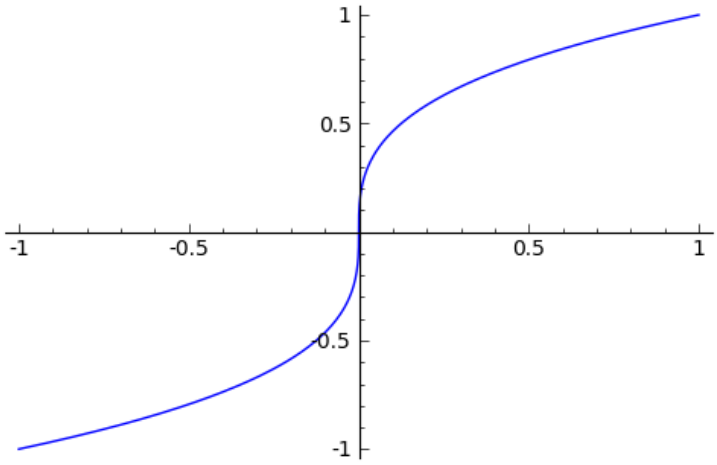
and not

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

which produces the error message we just discussed.

The code with the lambda produces the nice smooth beautiful cube root graph which we would expect to see.

Here is the plot:

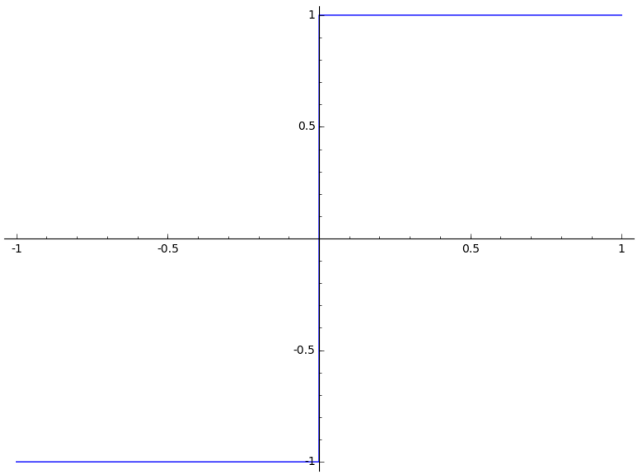


An alternative is to use

```
plot( sign(x) * (abs(x)^(1/3) ),  -1, 1)
```

which is a lot easier to explain. The absolute value, computed by `abs` will guarantee that the input to the one-third power is positive. However,

negative x-values should have negative cube roots, and positive x-values should have positive cube roots. Therefore, we multiply by `sign(x)`. The function `sign(x)` will return -1 for any negative x and $+1$ for any positive x, but it will return 0 for $x = 0$. By multiplying by `sign(x)` we guarantee that the computed cube root will have the correct sign. You can see a graph of the `sign(x)` function below:



A New Sage Function is Coming!

There are lots of reasons why Sage should have a command to get around this issue with the one-third power of negative numbers. For example, one might want to take derivatives and integrals of the cube root function.

With this in mind, just before this book went to the publisher, I started a discussion on the Google Groups `Sage-EDU` and `Sage-devel`, which are for Sage-related teaching and development topics, respectively. This is the beauty of open-source software that has been developed by a community. One can have conversations with the developers, make suggestions, ask for modifications, debate features, and eventually if a consensus forms, then changes happen. It is much harder to persuade a corporation to make changes to its commercial product, especially when there is no line of communication to the developers.

The new function is likely to be called `real_nth_root`, or something very similar, and will always provide the real cube root, fifth root, seventh root, or whatever you might want.

To Restrict the x-values of a Graph

This short discussion is not related to cube roots, but rather to square roots or any other function that has a domain other than the entire real line. When working with functions that have limited domains, once in a great while, you need to restrict the x-values of a graph.

Consider the function

$$f(x) = \sqrt{5x+8} + \log(3-9x) + \sqrt{1-4x}$$

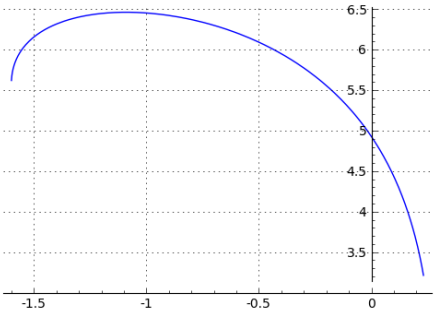
which clearly has a domain of $-8/5 \leq x \leq 1/4$. If you type

```
plot( sqrt(5*x+8) + log(3-9*x) + sqrt(1-4*x), -2, 2, gridlines=True)
```

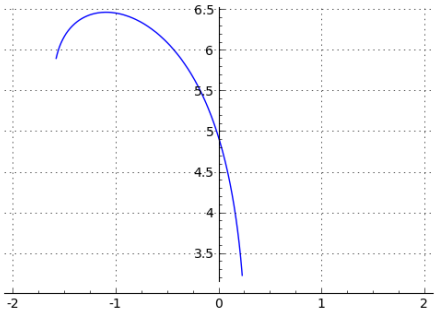
even though you asked for $-2 \leq x \leq 2$, you only get $-8/5 \leq x \leq 1/4$. It is also accompanied by an error message. Instead, you must do

```
plot( sqrt(5*x+8) + log(3-9*x) + sqrt(1-4*x), -2, 2,
          gridlines=True).show(xmin=-2,xmax=2)
```

where `xmin` and `xmax` explicitly force the x-coordinates to be what you want.



Plot using Show



Plot without Show

Chapter 4

Advanced Features of Sage

In this chapter will be found commands and examples for selected topics of mathematics that were not common enough to be given in Chapter One. Except in a few places (as noted) the sections are totally independent from each other, and certainly should not be read in order. Simply dive into sections of this chapter as you might happen to require one bit of advanced mathematics or another.

4.1. Using Sage with Multivariable Functions and Equations

Let's consider the following function

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

and see what Sage can do with it.

Just as we saw in Section 1.8.1 on Page 38, when we use variables other than x, unless they are constants like $k = 355/113$, we need to declare them as variables using the `var` command. For example

```
var('y')
f(x,y) = (x^4 - 5*x^2 + 4 + y^2)^2
f(1, 2)
```

The above code will define the function $f(x, y)$ as desired, and correctly output the value $f(1, 2) = 16$. From this, we can see that multivariate functions work the same way as univariate functions for definition and evaluation. However, we can also engage in half-evaluation, and get a function of fewer variables.

Just as one-variable functions are graphed on two-dimensional pieces of paper (or computer screens), you can graph two-variable functions in three dimensional space. This is covered in the electronic-only online appendix to this book "Plotting in Color, in 3D, and Animations," available on my webpage `www.gregorybard.com` for downloading.

We looked at parametric functions in Section 3.6 on Page 112. One thing you might want to be able to do is evaluate $f(x, y)$, as above, but with $x = 2t + 5$ and $y = 3t - 1$, or something similar. We would type

```
var('y t')
f(x,y) = (x^4 - 5*x^2 + 4 + y^2)^2
f(x=2*t+5, y=3*t-1)
```

and we receive the output

```
((2*t + 5)^4 + (3*t - 1)^2 - 5*(2*t + 5)^2 + 4)^2
```

which is clearly true but not very readable.

Alternatively, if we replace the last line with

```
f(x=2*t+5, y=3*t-1).expand()
```

then we'll get the more human-readable output

```
256*t^8 + 5120*t^7 + 44448*t^6 + 217088*t^5 + 649161*t^4
     + 1214732*t^3 + 1394126*t^2 + 902940*t + 255025
```

Half-Evaluating a Function:

Consider the following code:

```
var('y')
f(x,y) = (x^4 - 5*x^2 + 4 + y^2)^2
print f(x,y)
print f(x=1)
print f(y=2)
```

That code produces the following output:

```
(x^4 - 5*x^2 + y^2 + 4)^2
y^4
(x^4 - 5*x^2 + 8)^2
```

The first line is the actual function $f(x, y)$ as we had defined it earlier. The second line is what happens if x is locked at $x = 1$. In that case, we get a function of y alone, and it is a rather simple function. The third line is what happens if y is locked at $y = 2$. As you might guess, we get a function of x alone. This process of locking one or more of the variables of a function, while leaving one or more of the variables of a function free, is called "half evaluation."

Working with Parameters

A more extended example of "half-evaluation" is to take a function with four, five, or six variables, and plug in values for several of them (such as mass, thrust, $g = 9.82$, or other coefficients) in order to obtain an equation in fewer variables.

We'll see an example of that in Section 4.22.4 on Page 199.

4.2. Working with Large Formulas in Sage

Here we're going to explore using two large formulas in Sage—one from finance and one from physics. The finance example is a personal home mortgage. The physics example is Newton's Law of Gravitation. In both cases we're going to work with some complicated formulas and models, by writing a chunk of Sage code that eventually grows to about 4–6 lines long. This section assumes that you've read the previous section, Section 4.1, starting on Page 127.

4.2.1. Personal Finance: Mortgages

This example assumes you know how to do mortgage calculations. If you don't, then you might want to read a different example—or perhaps you'd like to learn about mortgages, in which case you should keep reading. The following notation, lovingly called "the alphabet soup" by business students, is fairly standard among finance textbooks, but with some variation.

- The length of the loan (or an investment) is t years.
- The nominal rate of interest per year is r. (Note, 5% is designated as $r = 0.05$.)
- The number of times per year that the interest is compounded is m. For example, quarterly is $m = 4$ and weekly is $m = 52$, while monthly is $m = 12$ and annually is $m = 1$.
- The number of compoundings over the life of the loan (or an investment) is $n = mt$.
- The interest rate per compounding period (instead of per year) is $i = r/m$. For example, 8% compounded quarterly has $i = 0.02$ while 6% compounded monthly is $i = 0.005$.
- For loans (or investments) with regularly spaced equal payments, the amount of that payment is c.
- For loans (or investments) with one initial payment only, that payment is called the principal and is denoted P.

Some Warmup Problems:

If you deposit \$ 1000 once, and leave it for three years at the rate of 4.8% compounded monthly, then $r = 0.048$, $m = 12$, $i = r/m = 0.004$, $t = 3$, $n = 36$, $P = 1000$. The formula for the amount that you have at the end is just the compound interest formula (which different books typeset slightly differently):

$$A = P\left(1 + \frac{r}{m}\right)^{mt} = P\,(1 + i)^n$$

The answer can be obtained in Sage by typing

```
(1000)*(1 + 0.048/12)^{(12*3)}
```

which is \$ 1154.55.

If you're saving for retirement, and want to deposit $ 100 per biweekly paycheck for 40 years, and invest at the rate of 9.1% compounded bimonthly. Since there are 52 weeks per year, the biweekly paychecks will come 26 times per year and $m = 26$. Then we have $t = 40$, $r = 0.091$, $i = 0.0035$, $n = 1040$, and $c = 100$. The correct formula in this case is the "future value of an annuity" or the "increasing annuity" formula and is given by

$$FV = c\frac{(1+i)^n - 1}{i} = c\frac{(1+r/m)^{mt} - 1}{r/m}$$

which we can evaluate with

```
(100)*(  (1 + 0.0035)^1040 - 1 ) / 0.0035
```

obtaining the rather lovely sum of $ 1,052,872.11.

Last but not least, if you want to get a 30-year home mortgage, at the rate of 5.4% compounded monthly, making house payments of $ 900 per month, then you would have $t = 30$, $m = 12$, $n = 360$, $r = 0.054$, $i = r/m = 0.054/12 = 0.0045$, and $c = 900$. The correct formula for this problem is the "present value of an annuity" or the "decreasing annuity" formula and is given by

$$PV = c\frac{1 - (1+i)^n}{i} = c\frac{1 - (1+r/m)^{mt}}{r/m}$$

which we can evaluate with

```
(900)*( 1 - (1 + 0.0045)^(-360) ) / 0.0045
```

obtaining the result of $ 160,276.16. Depending on where you live, that can be a very nice townhouse. For example, if you find a nice place with an asking price of $ 178,084.62, then putting 10% down would imply a down payment of $ 17,808.46, leaving $ 160,276.16 for the bank to provide via this mortgage.

Making a Cost-Per-Thousand Tabulator:

The three warm-up problems above might be easy for you, because you are fairly advanced in mathematics. Many bank employees are not college graduates at all, and are incapable of the preceding calculations. To enable such employees to be able to recruit customers and offer them mortgages, the "cost per thousand" technique is used.

A "rate sheet" is published by the bank either every day or every week. These sheets list a "cost per thousand" or "CPT" for each of the bank's loans. For example, there should be CPT for a 30-year fixed-rate mortgage, for a 15-year fixed-rate mortgage, and perhaps three CPTs for car loans (of 36, 48, and 60 months duration), followed by other loans (for boats, for education, and so forth).

Continuing with the example above, with 30-year mortgages and 5.4% compounded monthly, the CPT happens to be 5.6153079187. (I'll explain how to calculate that momentarily.) Because you understand mathematics,

if a customer came to you and asked how much the monthly payment would be for a $ 180,000 house, you'd be able to plug 180,000 into the PV equation above, plug in $n = 360$ and $i = 0.0045$, and just solve for c. However, the lowest layer of bank employees are unable to do that. Instead they will look up the CPT on the rate sheet, find that it is 5.6153079187, and just compute

$$(180)(5.6153079187) = 1010.75\cdots$$

enabling them to tell the customer that $ 1010.75 is going to be their monthly house payment with that mortgage.

Meanwhile, the rate sheet has to come from somewhere. A midlevel bank employee would compute the CPT while generating the "rate sheet" by taking

$$PV = c\frac{1-(1+i)^n}{i}$$

solving for c and obtaining

$$c = \frac{(PV)(i)}{1-(1+i)^n}$$

where one can plug in the i and n values. Naturally, $PV = 1000$. Let's compute this now with Sage.

Since $c = 1000$ is not going to change, we can put as the first line

```
c=1000
```

and then, we could define our own function by typing

```
CostPerThousand(i, n) = ( c * i ) / (1 - (1+i)^(-n) )
```

as our second line. Note, our setting $c = 1000$ here is not unlike how we had set $k = 355/113$ on Page 32.

Keeping those two lines in there, we can use the function repeatedly as our third line. For example, for a 30-year mortgage at 6.5% compounded monthly, we can type as our third line

```
CostPerThousand( 0.065/12, 30*12 )
```

and then if the rate rises to 7% we can replace that line with

```
CostPerThousand( 0.07/12, 30*12 )
```

and so on, and so forth.

Notice that the function name has to be only one word, but by varying the capitalization that way (which programmers call "Camel Case"), we could express an idea that was more than one word. Surely it is much easier to read `CostPerThousand` than `costperthousand` or `COSTPERTHOUSAND`.

While this function is nice, (for example it will save us from agonizing each time over the placement of the parentheses,) it also isn't quite perfect. Our data will be given in terms of the number of years of the loan t, not the number of payments $n = mt = 12t$. Likewise, the rate will be the nominal, published rate r not the per-period rate $i = r/m = r/12$. One student once described this to me by saying that the n and i are math-friendly, whereas the t and r are human-friendly.

This can be fixed by defining a new function. To avoid any confusion, our entire code chunk in the Sage Cell Server window should look like the following:

```
c=1000
CostPerThousand(i, n) = ( c * i ) / (1 - (1+i)^(-n) )
CostPerThousand1(r, t)=CostPerThousand( i=r/12, n=12*t )
```

Now we have made a new formula, which uses the rate in terms of r as we liked, and the number of years t, not the number of compounding periods. And now, for a 30 year mortgage at 6.5% compounded monthly, we can type

```
CostPerThousand1 ( 0.065, 30)
```

and get a more user-friendly input and output. One last reminder: when working with large sums, such as the cost of a house, we must be careful never to use a comma as a "thousands separator." All the digits must be consecutive, as we learned about on Page 2 and Page 3.

The Grouping Symbols:

Some older math books have the habit of using square brackets, [and], as a kind of super-parentheses. This was done to avoid having sets of parentheses inside of other parentheses. Someone raised on such books might be typed to write

$$f(i,n) = (c \cdot i) \div \left[1 - (1+i)^{-n}\right]$$

instead of the more modern

$$f(i,n) = (c \cdot i) \div \left(1 - (1+i)^{-n}\right)$$

which is identical except for the [becoming a (and the] becoming a).

Mathematicians can write what they'd like on paper, but in Sage code, there is much less flexibility. This is mostly because Sage is built on top of the computer language Python, and therefore Python conventions must be respected.

Therefore, one cannot type

```
CostPerThousand(i, n) = ( c * i ) / [1 - (1+i)^(-n) ]
```

but must instead type

```
CostPerThousand(i, n) = ( c * i ) / (1 - (1+i)^(-n) )
```

Remember, in Sage, the only grouping operators are parentheses. Additional explanation can be found on Page 2.

4.2.2. Physics: Gravitation and Satellites

Let's say you're analyzing a satellite in its orbit around the earth. Newton's formula for gravity is

$$F = \frac{GM_eM_s}{r_s^2}$$

where $G = 6.77 \times 10^{-11}\text{N m}^2/\text{kg}^2$ is the universal constant for gravitation; $M_e = 5.9742 \times 10^{24}$ kg is the mass of the earth; next r_s is the distance from the satellite to the center of the earth in meters; and finally M_s kg is the mass of the satellite.

Surely G and M_e will not change, so we can set those as the first two lines of our chunk of code by typing

```
G = 6.77 * 10^(-11)
Me = 5.9742 * 10^(24)
```

just like we had set $k = 355/113$ on Page 32.

Unlike the mass of the earth, or the universal constant of gravitation, the distance to the satellite might change somewhat often, and the mass of the satellite could change also (particularly if it has fuel aboard, which will be burned during maneuvers). Those two quantities should be variables and not constants. That means our force function will be a function of two variables. We would type as our third line:

```
Force( Ms, rs ) = (G * Me * Ms)/(rs^2)
```

Using these three lines, you can enter (on the fourth line) code such as:

```
Force( 1000, 10 * 10^6 )
```

to get answers. In this case, you are claiming that the mass of the satellite is 1000 kg and the distance from the satellite to the center of the earth is 10^7 meters; the answer returned should be 4044.53 Newtons. This notation is great if you have a large number of r values or M_s values that you're interested in, and want to save a lot of typing. You only need to change one number, or perhaps two, and just repeatedly click "Evaluate."

Suppose now that you really wanted the function to work in terms of altitude, as measured from the earth's surface, and not from the center of the earth. Well it is easy to realize that you just have to add the radius of the earth to the altitude to get the distance from the satellite to the center of the earth. First, you put in the radius of the earth, which is 6378.1 km, but of course we should switch the number to use meters, in order to avoid the error of conflicting units.

To avoid confusion, our entire code chunk should now look like

```
G = 6.77 * 10^(-11)
Me = 5.9742 * 10^(24)
re = 6378100
Force( Ms, rs ) = (G * Me * Ms)/(rs^2)
Force1( Ms, alt ) = Force( Ms, rs = alt + re )
```

The fifth line is very interesting. We've redefined r_s as the sum of the altitude and the radius of the earth. Also, note that we must never use commas as a "thousands separator" inside large numbers while programming in Sage—all the digits must be consecutive, as we learned about on Page 2 and Page 3. Now, we can just go ahead and use `Force1`. For example, as our sixth line, we can type

```
Force1( 1000, 3621900 )
```

which returns 4044.53 N, exactly what we had before. That's because an altitude of 3621.9 km is the same as being 10^7 meters from the center of the earth. This is true, in turn, because

$$3621.9 \text{ km} + 6378.1 \text{ km} = 10,000 \text{ km}$$

Which function should you use? `Force` or `Force1`? I suppose that it depends upon if the problem you are given is in terms of altitude, or in terms of r (the distance from the satellite to the center of the earth).

This idea of building functions on top of functions is very powerful. I have often written large suites of functions in this way—one function on top of the other function, on top of the next—to model complex phenomena in Sage. In this manner, the human brain need only consider one small matter at a time before moving on to the next small component, and therefore huge and complicated systems are rendered approachable and digestible.

4.3. Derivatives and Gradients in Multivariate Calculus

In Section 1.11 on Page 49, we learned about taking the derivatives of univariate functions. Now we'll learn about multivariate derivatives.

4.3.1. Partial Derivatives

You're probably not surprised to learn that Sage can do partial derivatives very easily. For example, if you have

$$g(x, y) = xy + \sin(x^2) + e^{-x}$$

you can type

```
g(x,y) = x*y + sin(x^2) + e^(-x)
diff(g(x,y), x)
```

and get

```
  2*x*cos(x^2) + y - e^(-x)
```

which is $\partial g/\partial x$. Similarly, you can type

```
g(x,y) = x*y + sin(x^2) + e^(-x)
derivative(g(x,y), y)
```

to learn that $\partial g/\partial y$ is just x. Note here this really shows how `diff` and `derivative` are synonyms.

You can even find

$$\frac{\partial^2}{\partial x \partial y} f$$

by typing `derivative( g(x,y), x, y)`. The answer, of course, is just 1.

4.3.2. Gradients

The next question we might be asked is to find the gradient of a multivariate function. If you want ∇f then you type

```
g(x,y) = x*y + sin(x^2) + e^(-x)
g.derivative()
```

and Sage will correctly calculate the gradient. Alternatively, you can type `diff(g)` in place of `g.derivative()`. In either case, Sage displays

```
(x, y) |--> (2*x*cos(x^2) + y - e^(-x), x)
```

which is to remind you that this is a map from a point (x, y) to a pair of real numbers, and not an ordinary function. Interestingly, the gradient of a univariate function such as $f(x) = 2e^{-x}$, done this way, is just the ordinary first derivative—which is exactly correct according to the laws of calculus.

4.4. Matrices in Sage, Part Two

Unlike the matrix section in Chapter 1 (see Page 19), this section does assume familiarity with the first semester of *Matrix Algebra* (sometimes called *Linear Algebra*). However, I have endeavored to make this section as readable and intuitive as possible. Those readers who are unfamiliar with matrix algebra, and who wish to acquire expertise, would do well to read Robert Beezer's *A First Course in Linear Algebra*, published electronically in 2007. Not only is that textbook entirely free, it is written with Sage in mind, and it is extremely readable.

4.4.1. Defining Some Examples

We're going to consider the following examples in this section. First, consider the very humble 3×3 matrix:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & -1 \end{bmatrix}$$

and also consider $\vec{b} =< 24, 63, 52 >$ as well as $\vec{c} =< 10, 14, 8 >$.

Note, that some books prefer to write $\vec{b} =< 24, 63, 52 >$ for a vector, and $B = (24, 63, 52)$ for a point in space. This distinction is an excellent one—useful to students and faculty alike—and mathematics books should use it. However, this notation is also a recent innovation, and might be unfamiliar to many readers who are accustomed to seeing parentheses used for both points and vectors.

Because Sage already uses the symbols `<` and `>` for comparisons, namely "less than" and "greater than," it is unfortunately necessary that Sage (like all but the most recent textbooks) cannot use the new notation. This is unavoidable. However, the `vector` command removes all confusion, as you will see momentarily.

By this point you know how to define A with

```
A = matrix( 3, 3, [1, 2, 3, 4, 5, 6, 7, 8, -1] )
```

as we learned on Page 22, but for $\vec{b}$ and $\vec{c}$ we have choices. We can consider $\vec{b}$ to be a matrix with 3 rows and 1 column. In that case we would write

```
b = matrix( 3, 1, [24, 63, 52] )
```

but there is also the vector command, which has the following syntax:

```
c = vector( [10, 14, 8] )
```

which requires less typing and is more human readable.

Take a moment in a Sage Cell Server to define these three objects, namely A, $\vec{b}$, and $\vec{c}$. Throughout the remainder of this entire section, just keep those three lines in the Sage cell. This way you do not have to retype them, and you can just change the last line of your cell as we learn various operations.

4.4.2. Matrix Multiplication and Exponentiation

The `*` symbol or asterisk is used to perform matrix multiplication just as it is used to perform the multiplication of rational numbers. For example, one can type `A*b` or `A*A` and get the desired answer. Similarly, the addition and the subtraction of matrices are done with the usual plus sign and minus sign, as if you were adding or subtracting numbers.

It is useful, at times, to have the identity matrix ready. You could define the identity matrix with

```
Id = matrix( 3, 3, [1, 0, 0, 0, 1, 0, 0, 0, 1] )
```

if you like. However, that can be tedious, especially for larger sizes. In particular, I often have trouble getting the right number of zeros to appear in the right places. Instead, you can define the identity matrix by using

```
Id = identity_matrix(3)
```

which is much shorter. Notice that I have the habit of calling the identity matrix `Id` and not `I`. This is merely for readability, as a single capital "I" can be mistaken for a lowercase L, or the number one, or the pipe above the backslash. You can call the identity matrix whatever you like. It is useful to see that `Id*A` and `A*Id` will produce the correct answers, namely A.

You can also raise matrices to various powers. The following code:

```
A = matrix( 3, 3, [1, 2, 3, 4, 5, 6, 7, 8, -1] )
print A*A*A
print
print A^3
```

demonstrates that Sage can perform the matrix multiplication much faster than a human. It also shows how matrix exponents use the same notation as the exponent of any number would.

In order to preserve the rule $(A^n)(A^m) = A^{n+m}$ from the real-number world, mathematicians define negative exponents with $A^{-n} = (A^{-1})^n$, provided that A is invertible. Also A^0 is the identity matrix. There is actually

a notion of the "square root" of a matrix, but it is rarely used and we will not go into it here.

4.4.3. Right and Left System Solving

Now let's use the objects defined earlier in this section to solve some problems. To find a vector $\vec{x}$ such that $A\vec{x} = \vec{b}$ we would type `A.solve_right(b)` and learn that the solution is $\vec{x} =< 7, 1, 5 >$. We can also verify this by typing

```
A*vector([7,1,5])
```

and we get the vector $< 24, 63, 52 >$ back, as desired.

To find a vector $\vec{y}$ such that $\vec{y}A = \vec{c}$ we would type `A.solve_left(c)` and learn that the solution is $\vec{y} =< 3, 0, 1 >$. As before, we would verify this by typing

```
vector([3,0,1])*A
```

and we get the vector $< 10, 14, 8 >$ back, as desired. There is a minor technical point here. The vector $< 7, 1, 5 >$ was a column vector, or a 3-row, 1-column matrix. The vector $< 3, 0, 1 >$ was a row vector, or a 1-row, 3-column matrix. Sage will always try to figure out if you meant a row vector or a column vector when using the `vector` command. However, some instructors might be less lenient, and require you to write instead $\vec{y}^T A$ in place of writing $\vec{y}A$.

The backslash operator can be used to abbreviate `solve_right`. Note that the backslash is not the one with the question mark, but the one with the pipe. (The one near the question mark is called the "forward slash" or solidus.) We would type

```
A \ b
```

as an abbreviation for `A.solve_right(b)`. This extremely concise abbreviation is a throwback to very old language `MATLAB`, which is still in use in industry, and which has the same backslash operator.

We can also type `A.augment(b)` to obtain the matrix

$$\left[\begin{array}{ccc|c} 1 & 2 & 3 & 24 \\ 4 & 5 & 6 & 63 \\ 7 & 8 & -1 & 52 \end{array}\right]$$

in preparation for using the `rref` command that we learned on Page 22. As we saw earlier, Sage does not draw the vertical line to mark the last column of the matrix in a special way. Some textbooks leave that vertical line out as well. To take the augment-and-RREF strategy, we would type

```
A = matrix( 3, 3, [1, 2, 3, 4, 5, 6, 7, 8, -1] )

b = matrix( 3, 1, [24, 63, 52] )

M = A.augment(b)
```

```
print "Before:"
print M
print
print "After:"
print M.rref()
```

Of course, the `print` commands aren't really necessary but they are very informative to the user. Some mathematicians will use the backslash operator or the `solve_right` command. However, this augment-and-RREF strategy does have its advantages.

Consider changing the last entry in A, namely A_{33} to 9 instead of -1. Then run our augment-and-RREF code again.

We see the following output

```
Before:
[ 1  2  3 24]
[ 4  5  6 63]
[ 7  8  9 52]

After:
[ 1  0 -1  0]
[ 0  1  2  0]
[ 0  0  0  1]
```

This is extremely informative. From the row of zeros followed by a 1, we see that the original system of equations has no solution. We can also see that the first and second column are effective, or elementary vectors, while the third column is defective, or a non-elementary vector. If you know what "rank" means in linear algebra, we can see that the matrix has rank 2. If you don't know what that means, do not worry about it right now. Let's see what would happen if we were to put the same question to the backslash operator.

If we were to type

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

b = matrix( 3, 1, [24, 63, 52] )

A \ b
```

we would get the response:

```
ValueError: matrix equation has no solutions
```

which is undeniably true but not as informative as the augment-and-RREF strategy.

What happens if you use the backslash when there are infinitely many solutions? Let's try it and find out! The commands:

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

```
d = vector( [6, 15, 24] )

A \ d
```

produce the vector $< 0, 3, 0 >$, which is a perfectly valid solution. On the basis of this output we would be justified in assuming that this is the unique solution to that problem. However, we can also see that $< 1, 1, 1 >$ is a solution. When there are infinitely many solutions, the backslash operator simply picks one. In particular, it will pick the solution formed by making all the free variables equal to zero.

We could instead type

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

d = vector( [6, 15, 24] )

B = A.augment(d)

B.rref()
```

to get the output

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

which, if you know how to read it, is much more informative. The circumstance of having an infinite number of solutions to a linear system (in other words, having a free variable) is discussed in Section D on Page 309.

Last but not least, we must recall that `A.rref()` is asking for a "reduced row-echelon form." However, there is also the "row-echelon form" or REF. This half-completed version of the RREF is extremely useful to know about if you have to work with matrices by hand, but in the computer age, its role is secondary at best. The command to compute the REF instead of the RREF is simply

```
A.echelon_form()
```

4.4.4. Matrix Inverses

Some matrices have inverses, and some do not. We can compute a matrix inverse with `A.inverse()`. For example,

```
A = matrix( 3, 3, [1, 2, 3, 4, 5, 6, 7, 8, -1] )
B = A.inverse()
print B
```

produces the output

```
[-53/30  13/15  -1/10]
[ 23/15 -11/15    1/5]
[ -1/10    1/5  -1/10]
```

You can verify the correctness of the inverse by asking for `A*B` or alternatively `B*A`, both of which will return the identity matrix. You can also replace `A.inverse()` with A^{-1}, as below:

```
A = matrix( 3, 3, [1, 2, 3, 4, 5, 6, 7, 8, -1] )
B = A^(-1)
print B
```

You might be wondering what happens if you invert a singular matrix? We can simply change the -1 in the lower right-hand corner of the matrix into a 9. Then the matrix is singular. We get an error message consisting of a lot of garbage, followed by

```
ZeroDivisionError: input matrix must be nonsingular
```

Sometimes it is useful to compute the inverse of a matrix "the long way." To accomplish this, we attach an identity matrix of the appropriate size, to the right of the original matrix. For example, the code

```
A = matrix( 3, 3, [1, 2, 3, 4, 5, 6, 7, 8, 9] )
B = identity_matrix(3)
C = A.augment(B)

print "Before:"
print C

print "After:"
print C.rref()
```

produces the output below.

```
Before:
[1 2 3 1 0 0]
[4 5 6 0 1 0]
[7 8 9 0 0 1]

After:
[   1    0   -1    0 -8/3  5/3]
[   0    1    2    0  7/3 -4/3]
[   0    0    0    1   -2    1]
```

As you can see, the matrix is not invertible. The RREF-process got "stuck" in the third column. This is much more informative than the error message we would get if we tried to just invert A with `A.inverse()`.

4.4.5. Computing the Kernel of a Matrix

For any specific matrix A, there are a lot of times when it is useful to know what set of vectors $\vec{v}$ will have $\vec{v}A = \vec{0}$. Likewise, there are lots of times when it is useful to know what set of vectors $\vec{v}$ will have $A\vec{v} = \vec{0}$. These two sets are called the "left kernel of A" and the "right kernel of A," respectively.

Let's explore an example:

```
A = matrix( 2, 2, [-2, 7, 0, 0] )

print A.left_kernel()

print "Test:"
print vector([0, 1])*A
print A*vector([0, 1])
```

produces the output:

```
Free module of degree 2 and rank 1 over Integer Ring
Echelon basis matrix:
[0 1]
Test:
(0, 0)
(7, 0)
```

It is noteworthy that our test code verifies that $\vec{v}A = \vec{0}$, as desired, but $A\vec{v} \neq \vec{0}$, as it turns out. This is to emphasize that the left kernel and right kernel are not the same sets.

I'd now like to explain the first two lines of the output above. As we'll see throughout these examples, the "degree 2" part refers to the dimension of the vectors in the kernel. The "rank" is the dimension of the space which gets sent to zero—this is usually called the "nullity" of A, but one should take care to differentiate between "left nullity" and "right nullity." It is important to emphasize that the rank of the left kernel, the rank of the right kernel, and the rank of A itself, can all be different.

For example, the matrix

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 0 \end{bmatrix}$$

has rank 2, but its left kernel is rank 0 (just the zero vector) and its right kernel is rank 3. You can find these numbers immediately with the functions `A.right_nullity()`, `A.left_nullity()`, and `A.rank()`, for any matrix A.

Meanwhile, the equivalent code using `right_kernel` would be as follows:

```
A = matrix( 2, 2, [-2, 7, 0, 0] )

print A.right_kernel()

print "Test:"
print vector([7, 2])*A
print A*vector([7, 2])
```

which produces the output

```
Free module of degree 2 and rank 1 over Integer Ring
Echelon basis matrix:
[7 2]
Test:
```

```
(-14, 49)
(0, 0)
```

Again, we see that $A\vec{v} = \vec{0}$, as desired, but that $\vec{v}A \neq \vec{0}$. For sure, the left kernel and right kernel are distinct mathematical objects. Sometimes the right kernel is called the "null space" of A. For that reason, if one says *the* kernel of A in mathematics, they usually mean the right kernel. However, in Sage, `A.kernel()` refers to the left kernel, which is extremely unusual. The wise programmer will be explicit and use `A.right_kernel()` and `A.left_kernel()` so that there can be no possible confusion.

By the way, I was discussing this once with Prof. William Stein, and he specifically asked me to mention in the book that "real programmers" always check the documentation. No one actually remembers fine details like the distinction between the left kernel and the right kernel. Even if you are confident that you remember which is which, it is better to check anyway (using the techniques of Section 1.10 on Page 47) to avoid the possibility of being mistaken.

To show how the concept works more powerfully, I wrote a program for you. See Figure 1 on Page 143. Here you see a larger matrix and we compute the left kernel and right kernel very carefully. You can also see that if $\vec{a}$ and $\vec{b}$ are in the kernel, then $r_1\vec{a} + r_2\vec{b}$ will also be in the kernel, for any real numbers r_1 and r_2 that you might want to choose.

Applications of Computing Kernels

We just went through a lot of effort to compute the left kernel and right kernel of a matrix. Perhaps you might be curious what this is used for. There are several uses in pure mathematics. In applied mathematics, one use has to do with linear systems of equations with infinitely many solutions.

Another use, from cryptography, is the Quadratic Sieve algorithm for factoring the product of two large prime numbers. This is a method of attacking the RSA cryptosystem. As it comes to pass, I discuss the Quadratic Sieve extensively in Chapter 10 of another book that I've written, *Algebraic Cryptanalysis*, published by Springer in 2009. While most of that book requires knowledge from a previous course in cryptography, the chapter on the Quadratic Sieve is somewhat independent and would be readable by anyone who has taken a course in *Number Theory* or perhaps instead two semesters of *Abstract Algebra*, sometimes called *Modern Algebra*.

4.4.6. Determinants

Let's return briefly to the matrix that we opened this lesson with. If we wanted to know its determinant, all we would need to do is type is the following:

```
A = matrix( 3, 3, [1, 2, 3, 4, 5, 6, 7, 8, -1] )
det(A)
```

Left Kernel Code

```
A = matrix( 4, 4, [1, 2, 3, 0, 4, 5, 6, -1,
        7, 8, 9, -2, 12, 15, 18, -3] )

print "Input Matrix:"
print A
print

print "Nullspace:"
print A.left_kernel()
print

xvec = vector( [1, 1, 1, -1] )
yvec = vector( [0, 3, 0, -1] )

print "Tests:"
print xvec, "maps to", xvec*A
print yvec, "maps to", yvec*A
print (xvec+yvec), "maps to", (xvec+yvec)*A
print (xvec-2*yvec), "maps to", (xvec-2*yvec)*A
print (17*xvec + 47*yvec), "maps to",
print (17*xvec + 47*yvec)*A
print (71*xvec + 888*yvec), "maps to",
print (71*xvec + 888*yvec)*A
```

Right Kernel Code

```
A = matrix( 4, 4, [1, 2, 3, 0, 4, 5, 6, -1,
         7, 8, 9, -2, 12, 15, 18, -3] )

print "Input Matrix:"
print A
print

print "Nullspace:"
print A.right_kernel()
print

xvec = vector( [1, 1, -1, 3] )
yvec = vector( [0, 3, -2, 3] )

print "Tests:"
print xvec, "maps to", A*xvec
print yvec, "maps to", A*yvec
print (xvec+yvec), "maps to", A*(xvec+yvec)
print (xvec-2*yvec), "maps to", A*(xvec-2*yvec)
print (17*xvec + 47*yvec), "maps to",
print A*(17*xvec + 47*yvec)
print (71*xvec + 888*yvec), "maps to",
print A*(71*xvec + 888*yvec)
```

Left Kernel Output

```
%%% left_kernel
Input Matrix:
[ 1  2  3  0]
[ 4  5  6 -1]
[ 7  8  9 -2]
[12 15 18 -3]

Nullspace:
Free module of degree 4 and rank 2 over Integer Ring
Echelon basis matrix:
[ 1  1  1 -1]
[ 0  3  0 -1]

Tests:
(1, 1, 1, -1) maps to (0, 0, 0, 0)
(0, 3, 0, -1) maps to (0, 0, 0, 0)
(1, 4, 1, -2) maps to (0, 0, 0, 0)
(1, -5, 1, 1) maps to (0, 0, 0, 0)
(17, 158, 17, -64) maps to (0, 0, 0, 0)
(71, 2735, 71, -959) maps to (0, 0, 0, 0)
```

Right Kernel Output

```
 %%% this is using "right_kernel"
Input Matrix:
[ 1  2  3  0]
[ 4  5  6 -1]
[ 7  8  9 -2]
[12 15 18 -3]

Nullspace:
Free module of degree 4 and rank 2 over Integer Ring
Echelon basis matrix:
[ 1  1 -1  3]
[ 0  3 -2  3]

Tests:
(1, 1, -1, 3) maps to (0, 0, 0, 0)
(0, 3, -2, 3) maps to (0, 0, 0, 0)
(1, 4, -3, 6) maps to (0, 0, 0, 0)
(1, -5, 3, -3) maps to (0, 0, 0, 0)
(17, 158, -111, 192) maps to (0, 0, 0, 0)
(71, 2735, -1847, 2877) maps to (0, 0, 0, 0)
```

FIGURE 1. The very large example for the `left_kernel` and `right_kernel` commands.

Then we would see that the determinant is 30. If we change the -1 to a 9, then the determinant will become 0. (Don't forget to hit "Evaluate" after making that change.) This makes sense, because we saw that A had a unique solution when we tried system solving with the vector $< 24, 63, 52 >$, and the -1 still present. On the other hand, we saw that `A \ b` had no solutions

Operation	Math Notation	Sage Syntax	Expected Answer
Addition	$\vec{a}+\vec{b}$	`a + b`	$<5,7,9>$
Subtraction	$\vec{a}-\vec{b}$	`a - b`	$<-3,-3,-3>$
Scalar Product	$3\vec{a}$	`3*a`	$<3,6,9>$
Dot Product	$\vec{a}\cdot\vec{b}$	`a.dot_product(b)`	32
Cross Product	$\vec{a}\times\vec{b}$	`a.cross_product(b)`	$<-3,6,-3>$
Norm or Length	$\|\|\vec{a}\|\|$	`norm(a)`	$\sqrt{14}$

TABLE 1. The Basic Vector Operations in Sage

when we tried system solving with that same vector, but after changing the -1 into a 9.

The cases of "infinitely many solutions" and "no solutions," can and will occur only with determinant equal to zero. The case of a unique solution is what occurs if and only if the determinant is non-zero.

4.5. Vector Operations

Vectors come up in all sorts of classes, including *Linear Algebra* and *Multivariate Calculus*, as well as *Physics I* and its successors. Sage is very good at working with vectors. Here we will explore the basic operations of adding, subtracting, taking a scalar multiple, computing a dot product, computing a cross product, and measuring the norm of a vector. Then we will combine them by exploring how to compute the angle between two vectors. We'll use, as our running example, the following two vectors:

$$\vec{a} =< 1,2,3 > \text{ and } \vec{b} =< 4,5,6 >$$

Each of the commands in Table 1 is intended to be prefaced by the definitions of these vectors, namely via:

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

For an example of how to combine several of these to perform a useful task, let's find the angle between $\vec{a}$ and $\vec{b}$. We must recall the formula

$$\cos\theta = \frac{\vec{a}\cdot\vec{b}}{||\vec{a}||\ \ ||\vec{b}||}$$

and then we can type

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

dotproduct = a.dot_product(b)
myfraction = dotproduct / ( norm(a) * norm(b) )
theta = arccos(myfraction)
```

```
print N(theta), "radians"
print N(theta*180/pi), "degrees"
```

That will result in the following output:

```
0.225726128552734 radians
12.9331544918991 degrees
```

4.6. Working with the Integers and Number Theory

The study of problems that deal with the integers is called "number theory." It was a famously pure subject for centuries, but in the last fifty years some amazing applications have come up, including cryptography and integer programming.

This subject can be started at a very young age (e.g. "what is a prime number?") but can be very advanced too. Many Ph.D. dissertations are written every year with new results in number theory. Mostly, number theory is about writing proofs, and one might imagine that Sage is not very useful for proof writing. Actually, Sage is very good at working with the integers to help you work out specific examples on what the instruments of number theory do to specific integers. This can help you understand new concepts more intimately, and I find it can form a bridge between the lecture on a topic and the time when you are really ready to write serious proofs about a topic.

For example, you can type `factor(-2007)` to find the factorization of that integer, which is

```
-1 * 3^2 * 223
```

and likewise

```
factor(2008)
```

produces the answer

```
2^3 * 251
```

To find the next prime number after one million, you can type

```
next_prime( 10^6 )
```

and discover the answer is 1,000,003.

You can also test a number for primality. Consider `is_prime(101)` or alternatively `is_prime(102)`. Of course, Sage says `True` for 101, and `False` for 102. Last but not least, you can find all the primes in a certain interval. For example,

```
prime_range(1000,1100)
```

gives you all the prime numbers between 1000 and 1100.

4.6.1. The gcd and the lcm

To find the "greatest common denominator" or "gcd" of two numbers, perhaps 120 and 64, just type

```
gcd ( 120, 64 )
```

and you will learn that the answer is 8. This makes sense because $120/8 = 15$ while $64/8 = 8$, and as you can see, 15 and 8 share no common factors. More explicitly, we can look at the prime factorizations $15 = 5 \times 3$ and $8 = 2 \times 2 \times 2$. Another way of looking at that is

$$120 = 2^3 \times 3 \times 5 \text{ while } 64 = 2^6$$

and, as you can see, all that 120 and 64 have in common is just 2^3, which equals 8. Therefore, the gcd is 8.

Likewise, you can find the "least common multiple" or lcm. This is done by typing

```
lcm( 120, 64 )
```

from which you learn that the lcm is 960. This makes sense because of the ratios $960/120 = 8$ and also $960/64 = 15$, and again 8 shares no factors with 15. A mathematician would say "8 is coprime to 15." Some authors say "mutually prime" instead of "coprime."

Another way to look at it is that the lcm should be

$$2^6 \times 3^1 \times 5^1 = 960$$

because the 2 appears 6 times in 64 and 3 times in 120 (thus a maximum of 6), while the 3 and 5 both appear 0 times in 64 and 1 time in 120 (thus a maximum of 1 each). Raising each of those primes (2, 3, and 5) to those powers (6, 1, and 1) gives the lcm.

Before we continue, one more example is helpful. Let's work with slightly larger numbers, perhaps 3600 and 1000. First, we do the prime factorizations

$$3600 = 2^4 \times 3^2 \times 5^2 \text{ while } 1000 = 2^3 \times 5^3$$

and we can compute the lcm by taking the maximum exponent each time

$$2^4 \times 3^2 \times 5^3 = 18,000$$

and the gcd by taking the minimum exponent each time

$$2^3 \times 3^0 \times 5^2 = 200$$

Take a moment to verify that 200 goes into both 1000 and 3600, as well as the fact that no higher number will. Next, take another moment to verify that both 1000 and 3600 go into 18,000, but there is no lower number for which that's true. And of course, the product of the numbers divided by the gcd is the lcm, and the product of the numbers divided by the lcm is the gcd. A key idea here is to realize that the prime factorization of a number communicates an enormous amount of information about that number.

If you want to take the gcd of many numbers at once (for example 120, 55, 25, and 35), there's no need to do

```
gcd( 120, gcd(55, gcd(25, 35 ) ) )
```

because you can do instead

```
gcd( [ 120, 55, 25, 35 ] )
```

to get 5 and likewise

```
lcm( [ 120, 55, 25, 35 ] )
```

to get 46,200. This is another example of a list in Sage. You can enclose any data with `[` and `]`, and separate the entries with commas, to make a list. This is notation that Sage inherited from the computer language called Python.

4.6.2. More about Prime Numbers

One neat trick is that if you wanted to find the 54,321th prime number, you could type

```
nth_prime(54321)
```

and discover that it is 670,177. You can double check this with

```
factor(670177)
```

which returns itself, confirming that the number is prime.

One of the most fascinating things in mathematics is the distribution of the prime numbers. For example, you might want to see how the function "`nth_prime`" grows as x grows. In plainer English, how large is the millionth prime? the billionth prime? or the trillionth prime?

It turns out that a good approximation for the size of the xth prime is

$$f(x) = 1.13x \log_e x$$

but there are better approximations, also.

We can test this with

```
f(x) = 1.13*x*log(x)

print f(10^6)
print nth_prime(10^6)

print f(10^7)
print nth_prime(10^7)
```

That code produces produces the output

```
1.56115269304996e7
15485863
1.82134480855829e8
179424673
```

and after adjusting for the scientific notation, we can see that the estimate and reality are within about 1% of each other.

4.6.3. About Euler's Phi Function

Sometimes it is useful to know how many integers are coprime to x, from the range 1 to x. For example, how many numbers from 1 to 10 share a factor with 10, and how many are coprime with 10.

The prime divisors of 10 are 2 and 5, and thus any number whose prime factorization contains 2s and 5s will share a factor with 10. These include

$$\{2, 4, 5, 6, 8, 10\}$$

and therefore the other numbers

$$\{1, 3, 7, 9\}$$

which have no 2s and no 5s in their factorizations at all, are coprime with 10. In particular, if you know what modular arithmetic is, then these are the numbers that have "multiplicative inverses" or "reciprocals" mod 10. Since 4 numbers are coprime to 10, we will write $\phi(10) = 4$.

In general, there are $\phi(n)$ numbers z with $1 \leq z \leq n$, such that z is coprime to n. This is kind of an inside-out way of defining a mathematical function, but it turns out that ϕ is extremely useful when working with modular arithmetic, and in especially in cryptography.

Even if you don't know what modular arithmetic is, consider 7. Each of

$$\{1, 2, 3, 4, 5, 6\}$$

is coprime to 7, because the only prime that divides 7 is 7, and 7 does not divide any of those. So, 6 out of the 6 numbers are coprime to 7, and we would then write $\phi(7) = 6$.

Actually, it will always be the case for a prime number that $\phi(p) = p - 1$. On the other hand, consider 12. Then among

$$\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}$$

we see that

- both 2 and 12 are divisible by 2;
- both 3 and 12 are divisible by 3;
- both 4 and 12 are divisible by 2;
- both 6 and 12 are divisible by either 2 or 3;
- both 8 and 12 are divisible by 2;
- both 9 and 12 are divisible by 3;
- both 10 and 12 are divisible by 2.

For these reasons, we exclude 2, 3, 4, 6, 8, 9, and 10 from our list. Finally, this means that only 1, 5, 7, and 11 are coprime to 12, a mere four integers, and thus we write $\phi(12) = 4$.

This strategy of listing the numbers and checking, one at a time, is nice for small numbers, but it wouldn't be possible for a 100-digit number, such as when working with cryptography. However, Sage can compute $\phi(n)$ with a single command. All you need to do in Sage is type

```
euler_phi(12)
```

and then you learn that the answer is 4. It is called "`euler_phi`" because it was discovered by Leonhard Euler (1707—1783), who also discovered the number

$$e \approx 2.718281828\cdots$$

and many other mathematical concepts. He still holds the record for the largest number of published papers by a mathematician, even though he's been dead for more than 230 years.

A Fun Challenge:

As it comes to pass, if p and q are distinct prime numbers, the value of $\phi(pq)$ is very easy to calculate. However, I'm not going to tell you what the formula is. I think it would be great fun for you to just try a bunch of numbers of the form pq and see if you can discover the formula for yourself—it isn't too hard. Just remember that "distinct" primes implies $p \neq q$.

4.6.4. The Divisors of a Number

Meanwhile, consider the set of divisors of a number. If you want to know the set of divisors of 250, which means the set of positive integers dividing 250, you would type

```
divisors(250)
```

which results in the following list:

```
[1, 2, 5, 10, 25, 50, 125, 250]
```

Likewise, if you wanted to know the set of divisors of 312,500 then type

```
divisors(312500)
```

which gives the output

```
[1, 2, 4, 5, 10, 20, 25, 50, 100, 125, 250, 500, 625, 1250, 2500,
3125, 6250, 12500, 15625, 31250, 62500, 78125, 156250, 312500]
```

but remember, it is very important to not have any commas separating the thousands, when inputting large numbers into Sage. (This means that you cannot have a comma between the 2 and the 5 in 312,500.)

Just like ϕ, there are two fun number-theoretic functions that come up with divisors. The *size* of the set of divisors of n is denoted $\tau(n)$, where as the *sum* of the set of divisors of n is[1] denoted $\sigma(n)$. The symbol σ is the Greek letter "sigma" and the symbol τ is the Greek letter "tau." For example, we just saw the divisor list for 250. There were 8 divisors, and so $\tau(250) = 8$. On the other hand,

$$1 + 2 + 5 + 10 + 25 + 50 + 125 + 250 = 468$$

and thus $\sigma(250) = 468$. We can compute σ in Sage with `sigma(250)`. A memory hook to keep these from getting mixed up in your mind is that

[1] Be forewarned, some statisticians are offended the number theorists have high-jacked the symbol σ for this purpose, whereas throughout science, engineering, and statistics, σ is the symbol for the standard deviation.

sigma is the Greek letter equivalent to our "s," and "sum" begins with "s." However, there is no "s" in "tau" nor in "count." Likewise, both "tau" and "count" have a "t," but "sum" and "sigma" have no "t." Keep in mind, $\sigma(n)$ sums the divisors, while $\tau(n)$ counts the divisors.

Sometimes in number theory, we want to calculate the sums of the squares or cubes of the divisors. Consider that 45 has as its divisors

$$\text{divisors}(45) = \{1, 3, 5, 9, 15, 45\}$$

so we have sigma totaling to 78, because

$$1 + 3 + 5 + 9 + 15 + 45 = 78$$

Accordingly, the sum of squares would be

$$1^2 + 3^2 + 5^2 + 9^2 + 15^2 + 45^2 = 2366$$

and with cubes

$$1^3 + 3^3 + 5^3 + 9^3 + 15^3 + 45^3 = 95,382$$

but in Sage the commands for that are `sigma( 45, 2 )` for squares and `sigma( 45, 3 )` for cubes.

Likewise, `sigma( 45, 1 )` is the same as sigma. There are two ways to compute τ in Sage. One way is given by `sigma(3600, 0)` which just tabulates 1 for each divisor. In other words, each divisor is raised to the 0th power, and thus becomes 1. Then, if you add up all those 1s, you learn how many divisors there are.

Another way to compute $\tau(250)$ would be to type

```
len( divisors( 250 ) )
```

which returns the length of the list of the divisors of 250—well, that's exactly what τ means. The `len` keyword in Python will return the length of any list, regardless of the type of problem being attempted.

Amicable Pairs of Numbers

Related to this topic is an Arab tradition[2] from the medieval period. Consider two particular numbers 220 and 284. The divisors of 220 are

$$\text{divisors}(220) = \{1, 2, 4, 5, 10, 11, 20, 22, 44, 55, 110, 220\}$$

and then they would calculate

$$1 + 2 + 4 + 5 + 10 + 11 + 20 + 22 + 44 + 55 + 110 = 284$$

but on the other hand, the divisors of 284 are

$$\text{divisors}(284) = \{1, 2, 4, 71, 142, 284\}$$

[2]While the amicable numbers appeared in the writings of the Ancient Greeks, the first mathematician to write theorems about them was Al-Sabi Thabit ibn Qurra al Harrani (826–901), usually called Thebit or Thebith in Western European books, who was more known for astronomical and geographical work.

which sum up to

$$1 + 2 + 4 + 71 + 142 = 220$$

Therefore we see 220 and 284 are intimately related. The technical term is that 220 and 284 are an "amicable pair" of integers. The Arabs found this so moving that they inscribed these numbers on jewelry that they would give to their spouses, signifying an intimate relationship. These are called Amicable Pairs—220 representing the young lady, and 284 representing the young gentleman.

Observe that we omitted n when summing the divisors of n in this case. You can find this in Sage via

```
sigma( 220 )-220
```

and

```
sigma( 284 )-284
```

Two other examples of amicable pairs are 1184 and 1210, as well as 2620 and 2924.

4.6.5. Another Meaning for Tau

As you know, some fractions like 1/4 and 1/2 as well as 1/25 can be expressed as decimal fractions which terminate. Others, like 1/3 or 1/9 will repeat. It is an interesting theorem that a fraction can be expressed as an exact terminating decimal fraction of n decimal places (or fewer) if and only if the denominator divides 10^n. Take a moment to convince yourself of this, or alternatively, consider the following example. If we want to know what denominators result in a two-decimal-place (or fewer) exact terminating decimal, we would just need to see what positive integers divide 100. Those would be the permissible denominators. In Sage we'd type

```
divisors(100)
```

and then get

```
[1, 2, 4, 5, 10, 20, 25, 50, 100]
```

which are the only denominators which are given exactly as a terminating decimal with only one or two decimal places.

In binary, the analogous rule would be 2^n. It is obvious that the only positive integers which divide 2^n are the numbers $1, 2, 4, 8, \ldots, 2^n$. In general, for numbers "base b," the admissible denominators for fractions using two or fewer digits past the decimal point would be the divisors of b^2. So for binary, $2^2 = 4$, and the denominators are `[1, 2, 4]`.

As you can see, we can use this criterion for seeing which number bases are convenient for fractions, and which number bases are not. We can set ourselves up as judges, to compare the binary, decimal, Mayan (base 20) and Babylonian (base 60) systems.

Let's first consider the Babylonian numbering system which is base 60. Using `divisors(3600)`, we learn that two symbols (or fewer) would be sufficient to describe any fraction with the following denominators:

```
[1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18, 20, 24, 25, 30, 36,
40, 45, 48, 50, 60, 72, 75, 80, 90, 100, 120, 144, 150, 180, 200,
225, 240, 300, 360, 400, 450, 600, 720, 900, 1200, 1800, 3600]
```

For the Mayan system (base 20), we have a shorter list, obtained with `divisors(400)`.

```
[1, 2, 4, 5, 8, 10, 16, 20, 25, 40, 50, 80, 100, 200, 400]
```

As you can see, for each base, we now know how many admissible denominators exist.

- Base 2, there are 3 denominators. (Excluding 1, there are only 2.)
- Base 10, there are 9 denominators. (Excluding 1, there are only 8.)
- Base 20, there are 15 denominators. (Excluding 1, there are 14.)
- Base 60, there are 45 denominators. (Excluding 1, there are 44.)

Note: What we are really doing is computing $\tau(n^2)$.

As you can see, the Babylonian system is most convenient for simple division of products, livestock, and money during trade. Not only are there 44 ways to write fractions, exactly, using two symbols past the units place, but the list begins with

$$\{1, 2, 3, 4, 5, 6, 8, 9, 10, 12, \ldots\}$$

namely every possible denominator up to and including 12 with the exception of 7 and 11. Perhaps it is unsurprising that we use base 60 for angles in geometry and trigonometry as a result, as well as dividing hours into minutes and seconds.

4.6.6. Modular Arithmetic

To find out what 1939 is mod 37, you can type

```
1939 % 37
```

but the % operator takes the modular reduction only of the nearest number, unless parentheses are used. This means that to evalulate

$$(3^2)4 + 2 \bmod 5$$

you must not type

```
3^2*4 + 2 % 5
```

which evaluates to 38, but rather

```
(3^2*4 + 2) % 5
```

which gives 3, the correct answer.

The opposite of the "mod" operator is the integer quotient. To find the integer quotient of a number, use

```
25 // 4
```

to get 6. This is the quotient $25/4 = 6.25$ rounded down. More precisely, $x//y = z$ means that z is largest integer such $zy \leq x$. Since $(7)(4) = 28$, we can see that $(6)(4) = 24$ is the largest possible way to have $(z)(4) \leq 25$, with z being an integer.

4.6.7. Further Reading in Number Theory

For the reader who would like to further explore number theory, I have two books to recommend. The first is slightly easier: *An Introduction to Number Theory with Cryptography*, by James S. Kraft and Lawrence C. Washington, published in 2013 by Chapman & Hall.

The second is more Sage-focused, and has been written by the founder of Sage, Prof. William Stein. The title is *Elementary Number Theory: Primes, Congruences, and Secrets: A Computational Approach*, published by Springer under the series "Undergraduate Texts in Mathematics" in 2008.

4.7. Some Minor Commands of Sage

Here, we list a few commands that come up from time to time. They can be very useful in the situations that require them.

`min(x,y)`	Returns either x or y, whichever is smaller.
`max(x,y)`	Returns either x or y, whichever is greater.
`floor(x)`	Just round x down, or $\lfloor x \rfloor$
`ceil(x)`	Just round x up, or $\lceil x \rceil$
`factorial(n)`	This is $n!$, also called "n factorial."
`binomial(x,m)`	This is written $\binom{x}{m}$ or $\frac{x!}{m!(x-m)!}$ or ${}_xC_m$
`binomial(x,m) * m!`	This is usually written $\frac{x!}{m!}$ or ${}_xP_m$

4.7.1. Rounding, Floors, and Ceilings

As you can see, the `floor` command rounds a number down, and the `ceil` command rounds a number up.

In older books, the "floor" function is sometimes called "The Greatest Integer Function." This is because it can be defined by "the greatest integer that is less than or equal to x." Some older mathematics books will write $[x]$ instead of $\lfloor x \rfloor$. I suppose if you wanted, you could define the ceiling function by "the least integer that is greater than or equal to x."

4.7.2. Combinations and Permutations

In a lot of textbooks, particularly for *Finite Mathematics*, or *Quantitative Literacy*, as well as a basic *Probability & Statistics*, a lot of time is spent talking about combinations and permutations. For example, if I have 52 distinct cards in a deck of cards, I might ask how many 5-card hands I can build with them.

If order doesn't matter, such as in Poker, then I would have

$$ {}_{52}C_5 = \binom{52}{5} = \frac{52!}{5!\,47!} = \frac{52 \times 51 \times 50 \times 49 \times 48}{5 \times 4 \times 3 \times 2 \times 1} = 2,598,960 $$

possibilities, and Sage calls this `binomial(52, 5)`. This is an example of the "combinations" formula.

However, if order does matter, which is not the case for poker, then I would have

$$ {}_{52}P_5 = \binom{52}{5} 5! = \frac{52!}{47!} = 52 \times 51 \times 50 \times 49 \times 48 = 311,875,200 $$

possibilities. In Sage, you can type

```
factorial(52)/factorial(52-5)
```

for that, or you can type

```
binomial(52,5)*factorial(5)
```

both of which return the same answer. This is an example of the "permutations" formula.

You might be wondering why Sage does not have a command for permutations, when it does have a command for combinations. Well, there is a command for permutations, but it does something a bit more than the above. In the above work, we calculated *how many* permutations, but sometimes you might want to know what those permutations are.

For this you can type

```
S= Permutations(['ace', 'king', 'queen', 'jack'])
print "We expect ", len( S.list() ), " items below."
S.list()
```

and it will obediently display all 24 possibilities:

```
We expect  24  items below.

[['ace', 'king', 'queen', 'jack'],
 ['ace', 'king', 'jack', 'queen'],
 ['ace', 'queen', 'king', 'jack'],
 ['ace', 'queen', 'jack', 'king'],
 ['ace', 'jack', 'king', 'queen'],
 ['ace', 'jack', 'queen', 'king'],
 ['king', 'ace', 'queen', 'jack'],
 ['king', 'ace', 'jack', 'queen'],
 ['king', 'queen', 'ace', 'jack'],
 ['king', 'queen', 'jack', 'ace'],
 ['king', 'jack', 'ace', 'queen'],
 ['king', 'jack', 'queen', 'ace'],
 ['queen', 'ace', 'king', 'jack'],
 ['queen', 'ace', 'jack', 'king'],
 ['queen', 'king', 'ace', 'jack'],
 ['queen', 'king', 'jack', 'ace'],
 ['queen', 'jack', 'ace', 'king'],
 ['queen', 'jack', 'king', 'ace'],
 ['jack', 'ace', 'king', 'queen'],
```

Mathematician's Words	Sage functions
Hyperbolic Sine	`sinh(x)`
Hyperbolic Cosine	`cosh(x)`
Hyperbolic Tangent	`tanh(x)`
Hyperbolic Cotangent	`coth(x)`
Hyperbolic Secant	`sech(x)`
Hyperbolic Cosecant	`csch(x)`
Inverse Hyperbolic Sine	`arcsinh(x)` or `asinh(x)`
Inverse Hyperbolic Cosine	`arccosh(x)` or `acosh(x)`
Inverse Hyperbolic Tangent	`arctanh(x)` or `atanh(x)`
Inverse Hyperbolic Cotangent	`arccoth(x)` or `acoth(x)`
Inverse Hyperbolic Secant	`arcsech(x)` or `asech(x)`
Inverse Hyperbolic Cosecant	`arccsch(x)` or `acsch(x)`

TABLE 2. The Hyperbolic Trigonometric Functions and Their Inverses

```
 ['jack', 'ace', 'queen', 'king'],
 ['jack', 'king', 'ace', 'queen'],
 ['jack', 'king', 'queen', 'ace'],
 ['jack', 'queen', 'ace', 'king'],
 ['jack', 'queen', 'king', 'ace']]
```

Try instead

```
S= Permutations( [ 'ace', 'king', 'queen', 'jack', 'ten', 'nine' ] )
print "We expect ", len( S.list() ), " items below."
S.list()
```

if you are curious.

Actually, this is one of many uses of the "`Permutations`" infrastructure in Sage, but we cannot go into the others here, as it is a very advanced topic.

4.7.3. The Hyperbolic Trigonometric Functions

If you've learned the hyperbolic trigonometric functions, then that's great. Personally, I have never had them taught to me in a course (and I have a PhD), but I do know that they come in handy at times. They come up in physics mostly in the study of the catenary curve, which is the shape of a cable permitted to hang limp under its own weight, but attached at each end. Also, there are some integrals which are much more compact in their solution if you know about the hyperbolic functions. So don't feel bad if you've never seen them. These are the commands for all 12 of those functions, and they are given in Table 2.

4.8. Calculating Limits Expressly

If you wanted to calculate

$$\lim_{x\to 0} \frac{x-2}{x-3}$$

you should type

```
limit( (x-2)/(x-3), x=0)
```

to learn that the answer is 2/3, which is perhaps obvious. Perhaps less obvious would be the limit

$$\lim_{x\to 1} \frac{x^2-4x+3}{x^2-3x+2}$$

for which you should type

```
limit( (x^2-4*x+3)/(x^2-3*x+2), x=1)
```

to obtain the answer "2." If you do not know how to do that without a computer or calculator (in other words, only with pen and pencil), then let me suggest that you try factoring those two polynomials as a first step.

Last but not least, if you wanted to verify the strange and amazing fact

$$\lim_{x\to 0} (\cos x)^{1/x^2} = \frac{1}{\sqrt{e}}$$

then you should type

```
limit( cos(x)^(1/x^2), x=0)
```

to obtain the desired (but very surprising) answer

```
e^(-1/2)
```

To show that the last limit there, given in terms of cosine and x^2 is true, you could try values like $x = 0.001$ on any scientific calculator, or with Sage. But to prove the limit true is rather difficult. It requires two applications of L'Hôpital's Rule or a Taylor Series (but not both), so far as I am aware.

Infinite Limits:

Sometimes limits go to infinity. For example,

$$\lim_{x\to 0^+} \frac{1}{x^2} = \lim_{x\to 0^-} \frac{1}{x^2} = \lim_{x\to 0} \frac{1}{x^2} = \infty$$

is the classic example in most calculus textbooks. However, if we were to put $1/x^3$ in place of $1/x^2$ in those limits, then we'd have a problem.

If we approach 0 from the positive numbers, the limit for $1/x^3$ would be $+\infty$, but if we approach 0 from the negative numbers, the limit for $1/x^3$ would be $-\infty$. Therefore, the limit does not exist if we approach 0 in general.

To see how Sage handles this, consider the following code:

```
print limit( 1/x^2, x=0)
print limit( 1/x^3, x=0)
print limit( 1/x^4, x=0)
print limit( -1/x^4, x=0)
```

That code generates this output:

```
+Infinity
Infinity
+Infinity
-Infinity
```

As you can see, Sage will faithfully report `+Infinity` when the limit goes to $+\infty$, and it will faithfully report `-Infinity` when the limit goes to $-\infty$. The unsigned infinity is a bit surprising. We can further clarify by typing

```
print limit( 1/x^3, x=0, dir='right')
print limit( 1/x^3, x=0, dir='left')
print limit( 1/x^3, x=0 )
```

That code produces the following output:

```
+Infinity
-Infinity
Infinity
```

Essentially, Sage will report `Infinity` as the limit for $f(x)$ whenever

$$\lim_{x \to 0} |f(x)| = +\infty$$

provided that the conditions for reporting `+Infinity` or `-Infinity` are not also satisfied. Many calculus professors might be upset with this convention, however. Traditionally, the limit as $1/x^3$ goes to zero should be "*d.n.e.*," which is an abbreviation for "does not exist."

By the way `dir = 'right'` can be also described as `dir = '+'` and `dir = 'plus'`. Similarly, `dir = 'left'` can be also described as `dir = '-'` and `dir = 'minus'`. Different minds find one or another naming scheme to be more or less intuitive. However, it is very important to remember that the limit approaching from the negative side and the limit approaching from the positive side are (at times) not equal. This is a common trap on calculus exams.

4.9. Scatter Plots in Sage

Let's suppose you're examining some data given by the points

$$\{(0, 7.1); \quad (1, 5.2); \quad (2, 2.9); \quad (3, 1.05); \quad (4, -0.9)\}$$

First, you should tell Sage about the data using the following command, which is another example of how lists are written in Sage:

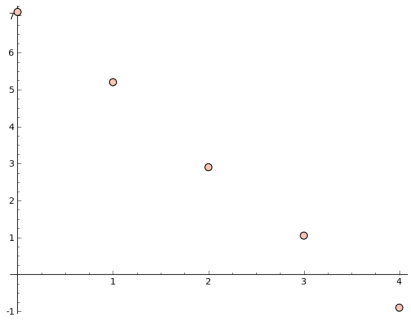
```
datapoints = [ (0, 7.1), (1, 5.2), (2, 2.9), (3, 1.05), (4, -0.9) ]
```

Notice how the data, namely those five 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 with commas, to make a list. This is notation that Sage inherited from the computer language called Python.

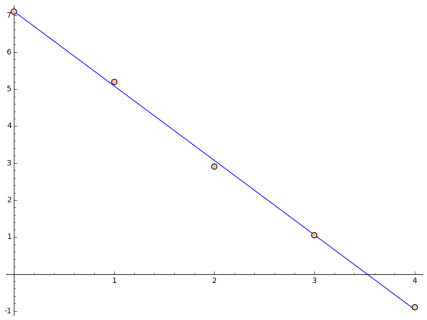
Now we can easily make a scatter plot, by just typing

```
scatter_plot( datapoints )
```

which will produce for us the scatter plot shown below on the left. However, it is important that the list with the data points remain inside of the Sage Cell server on your screen. So throughout this section and the next, you'll need that list of data points to be inside the box where you normally type commands.



Without Best-fit Line



With Best-fit Line

Perhaps you have the custom of computing "best fit lines" in a spreadsheet tool, or in a statistical package. If so, that's great. If not, then Sage will be very able to do this for you. We'll explain how to do that in the next section. Meanwhile, the actual best fit line for this example is $y = 7.1 - 2.015x$, and the scatter plot on the right includes this line. As you can see, it is a rather good fit, but an imperfect fit. That plot with the line included was created by typing

```
scatter_plot( datapoints ) + plot( 7.1-2.015*x, 0, 4 )
```

By the way, if you're curious how Sage came up with 7.1 and 2.015, that is discussed on Page 4.16.4. Now let's consider another data set. This example will explain to you why we are bothering to make a plot at all, when there are so many tools out there, including Sage, to give us the line of best fit whenever we want one. First, enter the data

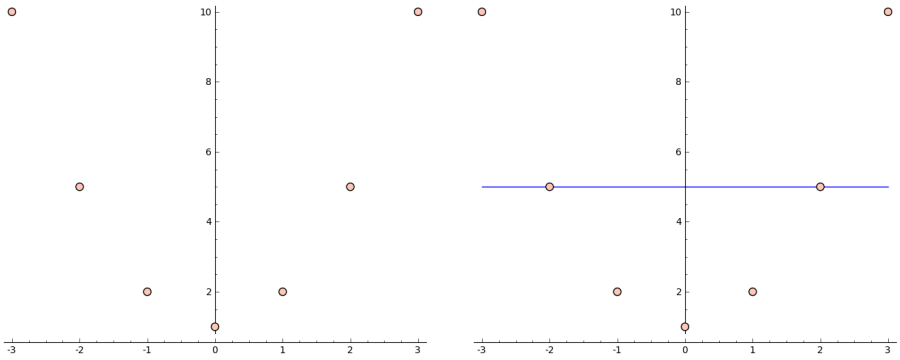
```
otherdata = [ (-3, 10), (-2, 5), (-1, 2), (0, 1),  (1, 2),
         (2, 5),  (3, 10) ]
```

and note that cut and paste usually works rather well if you don't want to retype the above list. Imagine that this is some data from a scientific experiment, and that you lab partner has copied the data and has found the line of best fit, $y = 0x + 5$ using some sort of statistics package, but

has not created a scatter plot. You, of course, are mindful of scientific best practices, and therefore you type

```
scatter_plot(otherdata)
```

to obtain the plot below on the left. Then you can add in the line of best fit, and you get the plot on the right in Figure 4.9.



Without Best-fit Line

With Best-fit Line

As you can see, this is absurd. The data is clearly an ordinary parabola. Therefore, a line is a terrible approximation for this data. However, of all the lines that could ever be drawn, even though all of them have unacceptably high error, there is one line that has the least error. It turns out that this line of least (squared) error, for this data set, is $y = 0x + 5$. Furthermore, it is manifestly clear that this line is a horrible fit, as shown in the plot on the right. It is for this reason that always plotting the scatter plot with the best fit line is considered to be a "standard operating procedure" in most labs. By the way, the previous plot was produced by the command:

```
scatter_plot(otherdata) + plot( 0*x+5, -3, 3)
```

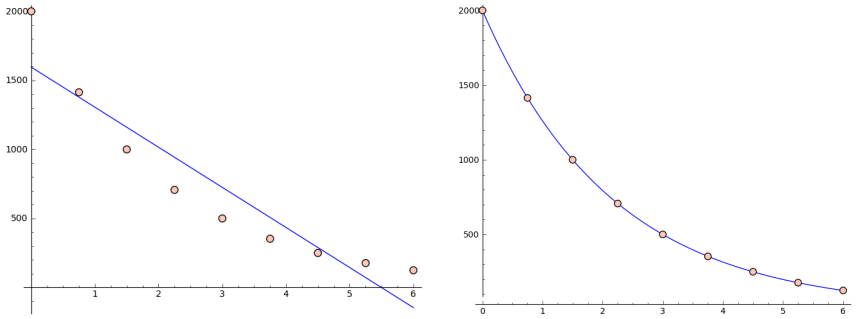
Now is a good time to explain that the object which scientists call "the best fit line" is actually better known among mathematicians as a "linear regression." By linear regression, we mean analyzing a set of data to find the line that best fits the data. Here, we should have done a "quadratic regression" which would find the parabola which best fits the data. In a few pages, we'll see how to do that.

A Chemistry Example:

Perhaps you might think that the parabola example is quite artificial. You're right, I did think it up just to prove a point. However, the following data shows how a 2 kilogram sample of Cobalt-62 might decay. Here the x-coordinate is time in minutes, and the y-coordinate is weight in grams.

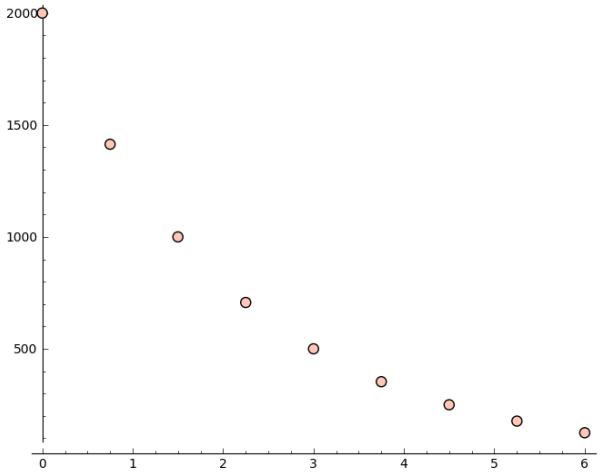
```
cobalt_data=[ (0, 2000), (0.75, 1414), (1.5, 1000), (2.25, 707),
    (3.00, 500),  (3.75,  353),   (4.50, 250),   (5.25, 177),
    (6.00, 125)]
```

We can type `scatter_plot(cobalt_data)` and get the scatter plot:



Linear Regression Exponential Regression

FIGURE 2. Two Plots of the Cobalt Data Set



An unwise student might be tempted to do a linear regression, especially if the last few lab reports required linear regressions. The linear regression of this data happens to be $y = -290.333x + 1596.11$. In that case, one gets the plot shown in Figure 4.9 on the left. However, a wiser student should remember that radioactive decay follows an exponential curve. Instead of asking for $y = ax+b$, which is a line, the wise student would ask for $y = ae^{bx}$, which is an exponential. This is part of the beauty of Sage's regression tools, namely that a linear regression is just as easy as a quadratic regression, an exponential regression, a cubic regression or a logarithmic regression. We'll see how to do those regressions in the next section. Meanwhile, the exponential regression happens to be $y = 1999.96e^{-0.462162x}$, and you get the curve shown below on the right.

The moral of the story is that while it is great to calculate regressions of data, you should not just assume that a linear regression is what you want. Dividing the world of functions into linear and non-linear functions is like dividing the set of living things on the earth into armadillos and

non-armadillos. Most functions are not linear, and therefore you should not assume that your data is best served by a linear model.

4.10. Making Your Own Regressions in Sage

This lesson assumes that you've read the previous section on scatter plots. In particular, we'll be using the cobalt data from the previous lesson. For the next few pages, you'll want that data as the first line of your Sage Cell window. If you are in SageMathCloud, this data just needs to be on the page and above the commands we are typing here.

First, we're going to try to fit a linear regression to that data. This means that we're going to ask Sage to give us a function $y = ax + b$ that models, as best as possible, these 9 points. The commands for that are

```
var("a b")
model(x) = a*x + b
find_fit(cobalt_data, model)
```

where the `var` command is one we've seen before (see Page 38), but the next two commands are new to us. The reply we get is

```
[a == -290.33333269040816, b == 1596.1111098318909]
```

which is Sage's way of telling us that the linear regression is

$$y = -290.333x + 1596.11$$

Now we can plot that with

```
scatter_plot(cobalt_data) + plot(-290.333*x+1596.11, 0,6)
```

Let's look at the plot, which is the left of the pair of images in Figure 4.9. As you can see, this is not a very good fit at all. Of course, basic chemistry tells us that radio-active decay follows an exponential curve. To do an exponential regression we type

```
var("a b")
model(x) = a*exp(b*x)
find_fit(cobalt_data, model)
```

and as you can see, only the middle line of those three lines has changed. We have changed the model from $y = ax + b$ into $y = ae^{bx}$. Now Sage's response to these commands is

```
[a == 1999.9607020042693, b == -0.46216218069391302]
```

which is Sage's way of telling us that the model is

$$y = 1999.96e^{-0.462162x}$$

and you can immediately see that there's some rounding error. That leading coefficient of 1999.96 really should be 2000, as we started with 2000 grams. However, that's part of the pros and cons of numerical methods. You can do almost anything with numerical methods, but you very rarely can be exact. In any case, we can see the quality of this regression by the command

```
scatter_plot(cobalt_data) + plot(1999.96*exp(-0.462162*x), 0, 6)
```

The plot obtained is the image on the right in Figure 4.9. Last but not least, one might consider a quadratic regression. That's a regression of the form $y = ax^2 + bx + c$, and the command for that would be

```
var("a b c")
model(x) = a*x^2 + b*x + c
find_fit(cobalt_data, model)
```

where you can see that the only real change is the middle of the three lines again, but also we had to declare c as a variable, which we did in the `var` command. In any case, the response from Sage is

```
[a == 62.755170737557854, b == -666.86435772164759,
         c == 1925.5757574133333]
```

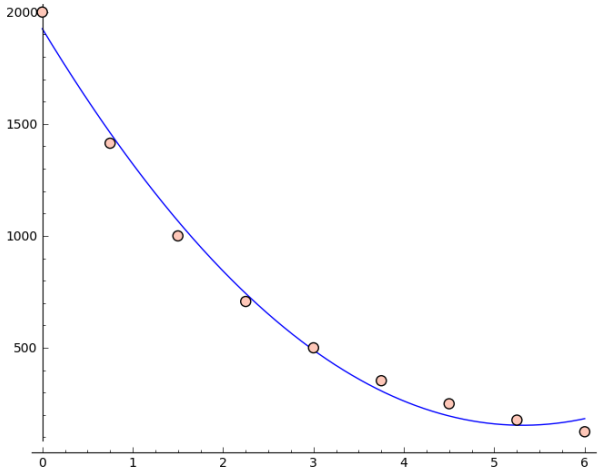
and that is Sage's way of telling us that the model is

$$y = 62.7551x^2 - 666.864x + 1925.57$$

Next, we can draw that model with

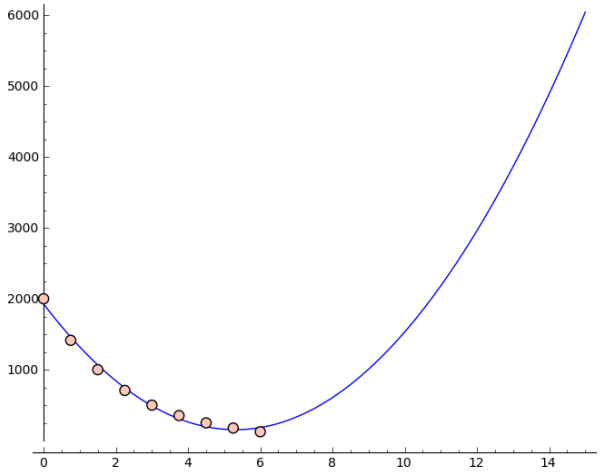
```
scatter_plot(cobalt_data) + plot(62.7551*x^2-666.864*x+1925.57,0,6)
```

which produces the image



While that's clearly not a very bad fit, it is unwise to model radioactive decay with a quadratic function. This is not only because many of the data points were "missed." The issue can be most easily seen by allowing time to have the range 0 to 15, instead of 0 to 6. Just change the 6 into a 15 at the end of the last line we just typed.

We get the following image



which is clearly *not* a good model of radioactive decay.

4.11. Computing in Octal? Binary? and Hexadecimal?

To find out the decimal values of hexadecimal 71, type `0x71` and get `113`. Note that this is because $7 \times 16 + 1 = 113$. By the way, that's a zero before the `x` not the letter "o." Alternatively, for hexadecimal `0x1F` you would type `0x1F`, and get 31, because $1 \times 16 + 15 = 31$.

The octal system is considerably more rare, but it does come up from time to time. For octal 11, type `011`. The response is 9 because $1 \times 8 + 1 = 9$.

Binary is similar to hexadecimal, except with a `b` in place of the `x`. Thus to find out what binary 1101 is, you would type `0b1101`, and learn that the answer is 13. That's because

$$(1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) = 8 + 4 + 0 + 1 = 13$$

To go in reverse, you can type `hex(31)` and you'll get `1F`. Likewise, for binary you can type `bin(63)` and you'll get the binary expansion for 63, which is 11111 (or five ones, consecutively). The octal-in-reverse is a bit messy. Type `oct(int(31))` to learn that the answer is `37`, because

$$37_{\text{octal}} = 3 \times 8 + 7 = 24 + 7 = 31$$

4.12. Can Sage do Sudoku?

To solve the 9×9 Sudoku puzzle defined by a 9×9 matrix, enter the matrix:

```
A = matrix( 9, 9, [5,0,0, 0,8,0, 0,4,9, 0,0,0, 5,0,0, 0,3,0,
   0,6,7, 3,0,0, 0,0,1, 1,5,0, 0,0,0, 0,0,0, 0,0,0, 2,0,8,
   0,0,0, 0,0,0, 0,0,0, 0,1,8, 7,0,0, 0,0,4, 1,5,0, 0,3,0,
   0,0,2, 0,0,0, 4,9,0, 0,5,0, 0,0,3])
```

Then if you type `print A` on a line by itself, you get

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

which has some non-zero entries, but mostly contains zeros to mark where the blanks would go in an ordinary Sudoku game. Now you can solve the game by typing

```
sudoku(A)
```

and obtain

```
[5 1 3 6 8 7 2 4 9]
[8 4 9 5 2 1 6 3 7]
[2 6 7 3 4 9 5 8 1]
[1 5 8 4 6 3 9 7 2]
[9 7 4 2 1 8 3 6 5]
[3 2 6 7 9 5 4 1 8]
[7 8 2 9 3 4 1 5 6]
[6 3 5 1 7 2 8 9 4]
[4 9 1 8 5 6 7 2 3]
```

which is a valid Sudoku solution. However, there might be other solutions.

4.13. Measuring the Speed of Sage

Factoring large numbers, for example, is supposed to be hard and you might want to know how long it takes Sage to execute the command

```
factor(2250000063000000041)
```

In which case you should type

```
timeit("factor(2250000063000000041)")
```

and for me, on my machine, it says

```
625 loops, best of 3:  91.8 µs per loop
```

which means it tried 625 times, and the best estimate for the running time is 91.8 microseconds, or about 1/10,893 seconds.

Of course, the response time when you operate with Sage is much longer than that. What's going on? Most of the time using Sage is wasted with the commands traveling over the internet to the server, and traveling back. Also, some time is wasted displaying the data. In this case, you probably lose about 2 seconds on transit time, 1/4th of a second on display and the interface, and nearly 0 seconds on computation.

However, for certain *very hard* problems, you can spend 10 minutes on computation, and 2 seconds on transit time, and a 1/4th of a second on display and the interface. So this feature of Sage is there to help you measure that.

4.14. Huge Numbers and Sage

Sometimes when studying combinatorics, one has to deal with numbers that are huge. For example, asking Sage for `factorial(52)` will result in

```
80658175170943878571660636856403766975289505440883277824000000000000
```

but it is hard for me to grasp what that means. However, I can type

```
floor(  log(   factorial(52), 10 )  )+1
```

which will give me 68. That means that the number is a 68-digit number, for reasons we will now explain.

Why does this work? Because $\log_{10} 10^7 = 7$ and $\log_{10} 10^8 = 8$. Accordingly, any number x such that $10^7 \leq x < 10^8$ has therefore $7 \leq \log_{10} x < 8$. However, what numbers have $10^7 \leq x < 10^8$? Precisely the eight-digit numbers of course! Therefore, the set of 8-digit numbers is precisely the set of numbers with $7 \leq \log_{10} x < 8$. By taking the floor with `floor`, we round down, and thus every 8-digit number has the floor of its common logarithm being precisely 7. We correct for this with a $+1$.

Typing `factorial(factorial(7))`, to compute (7!)! is also amusing. You get a very large number if you do this. In fact, using the above technique, I can learn that (7!)! has 16,474 digits. You'll also see that this very large number is displayed as dozens of lines each ending in a backslash. The backslashes are Sage's way of saying that it really sees several lines as one large line, but that the very large line of digits wouldn't fit on the screen. Therefore, Sage has to break it, to fit it on your screen.

It is also fun to type

```
factor( factorial( factorial( 7 ) ) )
```

I am impressed at how rapidly Sage can factor that 16,474-digit number. And observe that every single prime number less than 5040 is present in the factorization, though raised to various powers.

Fractions can be even more interesting. For example, if you type:

```
7200000*(21/20)^(2011-1867)
```

you get back

```
22585044527136650669987277163669962902773078414361940471413541987229799 1\
15989073012232547584871319909796580812726652370566841726773636316510045 0\
84408910795956242537828315632385661709050569952 9/27875931498163278926919\
64784081045188247552000000000000000000000000000000000000000000000000000\
00000000000000000000000000000000000000000000000000000000000000000000000\
000000000000000
```

Notice that there's a forward slash roughly 2/3rds of the way through the middle line. So that five-line number is really a 2.67-line integer divided

by a 2.33-line integer. More precisely, it is a 192-digit number divided by a 182-digit number. I discovered those digit counts by using the technique of taking the floor of the logarithm base 10, which we learned moments ago. In any case, typing

```
N( 7200000*(21/20)^(2011-1867) )
```

gives me the decimal approximation

```
8.10198738242156e9
```

and that's about 8.1 billion—something that I can wrap my head around, even if but partially.

In case you were wondering what that came from, look at the dates, and you'll quickly realize exactly what it is (if you've studied American History). In the year 1867, the USA purchased Alaska from Russia for 7.2 million dollars. If that sum had instead been invested at 5% compounded annually for the same period of time, it would be worth 8.1 billion dollars in 2011, when I first started this book. (Note that $21/20 = 1.05$.) Of course, that's an extremely moderate interest rate, and it displays the power of compound interest.

4.15. Using Sage and LaTeX

Perhaps I want to solve the following quartic equation

$$x^4 + x^3 + x^2 + x - 4 = 0$$

and I type the following code:

```
answers = solve(x^4 + x^3 + x^2 + x - 4, x)

print answers[0]
print answers[1]
print answers[2]
print answers[3]
```

Then I might see that `answers[2]` is the root that I am interested in. However, how am I going to efficiently typeset

```
x == (5/9*sqrt(3)*sqrt(2) - 35/27)^(1/3) -
         5/9/(5/9*sqrt(3)*sqrt(2) - 35/27)^(1/3) - 2/3
```

in LaTeX for a conference paper or a journal article? Luckily, Sage will do this for me. The following code:

```
answers = solve(x^4 + x^3 + x^2 + x - 4, x)

print answers[2]

latex(answers[2])
```

produces the following output

```
x == (5/9*sqrt(3)*sqrt(2) - 35/27)^(1/3) - 5/9/(5/9*sqrt(3)*sqrt(2)
      - 35/27)^(1/3) - 2/3

x={\left(\frac{5}{9} \, \sqrt{3} \sqrt{2} - \frac{35}{27}\right)}^{
      \frac{1}{3}} - \frac{5}{9 \, {\left(\frac{5}{9} \, \sqrt{3}
      \sqrt{2} - \frac{35}{27}\right)}^{\frac{1}{3}}}-\frac{2}{3}
```

As you can see, the first line is written in Sage and the second line is written in LaTeX. It produces the following very nice, but still imperfect, output.

$$x = \left(\frac{5}{9}\,\sqrt{3}\sqrt{2} - \frac{35}{27}\right)^{\frac{1}{3}} - \frac{5}{9\left(\frac{5}{9}\,\sqrt{3}\sqrt{2} - \frac{35}{27}\right)^{\frac{1}{3}}} - \frac{2}{3}$$

4.16. Matrices in Sage, Part Three

Large pieces of Sage use linear algebra to carry out their tasks, so it is unsurprising that Sage has a huge suite of commands dedicated to linear algebra. There is too much to go into here, but I have chosen three flagship topics to round out our third section on linear algebra. They are eigenvector/eigenvalue problems, matrix factorizations and canonical forms, as well as solving least-squares problems.

4.16.1. Introduction to Eigenvectors

A right-eigenvector $\vec{v} \neq \vec{0}$ of the matrix A is any vector such that there is some real or complex number λ with $A\vec{v} = \lambda\vec{v}$. In other words, when I multiply $A\vec{v}$, I get double $\vec{v}$, or one-third $\vec{v}$, or a million times $\vec{v}$, or some constant multiple of $\vec{v}$. Let us take it as given that these vectors are very important in certain scientific applications, and focus instead on finding them. By the way, the term for the λ that appears in the definition is "eigenvalue."

If A is an $n \times n$ matrix, then let I_n be the $n \times n$ identity matrix. The following formulations are equivalent:

$$\begin{aligned} A\vec{v} &= \lambda\vec{v} \\ A\vec{v} - \lambda\vec{v} &= \vec{0} \\ A\vec{v} - \lambda(I_n\vec{v}) &= \vec{0} \\ (A - \lambda(I_n))\vec{v} &= \vec{0} \end{aligned}$$

The last equation makes it clear that for any interesting (which is to say, non-zero) vectors $\vec{v}$ to be found, the matrix $(A - \lambda I_n)$ must be singular. Of course, that means that the determinant of $(A - \lambda I_n)$ is zero. We don't yet know what λ is, so let's call it x. We are essentially constructing a new matrix, where all the off-diagonal entries match those of A, but the on-diagonal entries of A have "$-x$" attached to them. We can ask Sage to take the determinant of that matrix, or compute the determinant by hand.

However, we know that if x is an eigenvalue, then that determinant is zero. Let's see a short example:

$$A = \begin{bmatrix} 0 & 0 & 0 & -4 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 5 \\ 0 & 0 & 1 & 0 \end{bmatrix} \quad \text{becomes } A - xI_n = \begin{bmatrix} 0-x & 0 & 0 & -4 \\ 1 & 0-x & 0 & 0 \\ 0 & 1 & 0-x & 5 \\ 0 & 0 & 1 & 0-x \end{bmatrix}$$

Next, it turns out the determinant is

$$\det(A - xI_n) = x^4 - 5x^2 + 4$$

which had me worried, because that's a fourth-degree polynomial. However, it is a biquadratic, so it factors easily

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

By the way, that polynomial we just found is called the characteristic polynomial of A. Now we know that the values of x that will be eigenvalues are

$$x \in \{-2, -1, 1, 2\}$$

and we can proceed to go find them.

Just taking $\lambda = 2$ as an example, we have to solve

$$(A - \lambda(I_n))\vec{v} = (A - 2I_n)\vec{v} = \vec{0}$$

but that's just a standard linear algebra problem. We learned how to do that on Page 137 using the backslash operator or the `solve_right` command. We have to do that for each of the 4 eigenvalues, but this is clearly not a difficult problem.

There are other ways to find the eigenvectors. What was convenient here is that the polynomial was easy to handle. Because of the fact that polynomials of degree 5 and higher are not guaranteed to be solvable, our previous technique (of factoring the characteristic polynomial) can be computationally infeasible. Luckily, Sage handles this for us.

In Sage, the entire process above can be streamlined, as below:

```
A = matrix([ [0,0,0,-4], [1,0,0,0], [0,1,0,5], [0,0,1,0] ])
I = identity_matrix(4)

print "A:"
print A
print
print "I:"
print I
print
print (A-I*x)
print
print det(A-I*x)
print factor(det(A-I*x))
```

This produces the output

```
A:
[ 0  0  0 -4]
[ 1  0  0  0]
[ 0  1  0  5]
[ 0  0  1  0]

I:
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]

[-x  0  0 -4]
[ 1 -x  0  0]
[ 0  1 -x  5]
[ 0  0  1 -x]

x^4 - 5*x^2 + 4
(x + 2)*(x + 1)*(x - 1)*(x - 2)
```

By the way, the characteristic polynomial of A has a fascinating relationship with A. If you type

```
print A^4 - 5*A^2 + 4
```

where we have taken the matrix A and treated it as if it were x, we get the zero matrix back—that will always occur.

4.16.2. Finding Eigenvalues Efficiently in Sage

The process in the previous section was to give you an idea about what finding an eigenvector or an eigenvalue actually means. The procedures used can be interesting both in the mathematics that they leverage and in the algorithmic computer science chosen to make the process run efficiently. We cannot go into that here, but I recommend the book by Lloyd Trefethen and David Bau, *Numerical Linear Algebra*, published by the Society for Industrial and Applied Mathematics in 1997.

We can find the eigenvalues or eigenvectors directly, with the commands:

```
A.eigenvalues()
A.eigenvectors_left()
A.eigenvectors_right()
```

However, sometimes the characteristic polynomial is valuable in its own right. To obtain it, we should use either of the following commands:

```
A.charpoly()
factor(A.charpoly())
```

To demonstrate the tradeoffs between these methods, we will investigate the following matrix:

$$B = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

Consider the following code:

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

print B.eigenvectors_right()
print
b(x) = B.charpoly()
print solve( b(x)==0, x )
```

That code produces the following output:

```
[(-0.3722813232690144?, [(1, -0.6861406616345072?)], 1),
          (5.372281323269015?, [(1, 2.186140661634508?)], 1)]

[
x == -1/2*sqrt(33) + 5/2,
x == 1/2*sqrt(33) + 5/2
]
```

We see a sequence of items separated by commas and enclosed in brackets, so we know that we have a list in the syntax of Sage and Python. Each of the two items in the list is an ordered triple, consisting of the eigenvalue λ, and the eigenvector $\vec{v}$, followed by the multiplicity of the eigenvalue in the matrix's characteristic polynomial.

Below that is the output of `solve` where we explicitly found the roots of the characteristic polynomial. Therefore, we obtain the exact, algebraic form of the eigenvalues. This is more pleasing to me, because otherwise, we don't really know what the significance of the numerical values might be. However, in many applications, the numerical values are all you need. Notice how Sage reminds you that the eigenvalues are numerical approximations in the `eigenvectors` command by use of the `?` symbol after the least-significant digit.

Earlier, I noted that if you "plug in" the matrix A in place of x inside its characteristic polynomial then you get the zero matrix back. The characteristic polynomial is usually, but not always, the lowest-degree polynomial that achieves this phenomenon. However, if the characteristic polynomial has repeated roots, there is another polynomial which is of lower degree, that will also send A to the zero matrix. That is the "minimum polynomial" of A, and it is found by the command `A.minpoly()`.

4.16.3. Matrix Factorizations

There are many canonical forms and factorizations of matrices in the world of mathematics. Sage has fourteen of them built in, that I could find. (There

might be more that I did not find.) If you have a particular application in mind that often uses one particular decomposition, factorization, or canonical form, then I hope that you will find it below.

```
A.as_sum_of_permutations()  A.LU()
A.cholesky()                A.QR()
A.frobenius()               A.rational_form()
A.gram_schmidt()            A.smith_form()
A.hessenberg_form()         A.symplectic_form()
A.hermite_form()            A.weak_popov_form()
A.jordan_form()             A.zigzag_form()
```

However, the most famous matrix factorization of all is the "LU-factorization." Usually, when someone speaks of the "LU-factorization" they mean the LUP factorization, where $A = LUP$, such that L is some lower-triangular matrix, U is some upper-triangular matrix, and P is some permutation matrix. That permutation matrix often, but not always, is the identity matrix. This tends to make it "optional" in the sense that most A can be written as $A = LU$ without need of $P \neq I_n$. In Sage, however, `A.LU()` is a PLU factorization. The permutation matrix is on the left, not the right, which is unusual. You are forewarned!

This did cause me some embarrassment once, in the Spring semester of 2014, when I was teaching *Numerical Analysis II*, and my factorization of $A = PLU$ did not multiply back to the original A. That's because I was using $A = LUP$.

4.16.4. Solving Linear Systems Approximately with Least Squares

What were to happen if I were to want to solve a system of five equations in two unknowns? It even sounds strange to the human ear—why are there five equations if there are only two unknowns? Perhaps I want to solve the following linear system of equations:

$$\begin{aligned} 0m + b &= 7.1 \\ 1m + b &= 5.2 \\ 2m + b &= 2.9 \\ 3m + b &= 1.05 \\ 4m + b &= -0.9 \end{aligned}$$

Because those equations are linear, we know that I can make this into a matrix problem:

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \\ 2 & 1 \\ 3 & 1 \\ 4 & 1 \end{bmatrix} \begin{bmatrix} m \\ b \end{bmatrix} = \begin{bmatrix} 7.1 \\ 5.2 \\ 2.9 \\ 1.05 \\ -0.9 \end{bmatrix}$$

Therefore, in Sage, I should type

```
A = matrix(5, 2, [0, 1, 1, 1, 2, 1, 3, 1, 4, 1] )
print A
print

b = matrix(5, 1, [7.1, 5.2, 2.9, 1.05, -0.9] )
print b

A \ b
```

Unsurprisingly, Sage responds that there is no solution. I have too many constraints, and not enough variables that I can change to be able to satisfy all five equations simultaneously. What, then, should I do?

What if an approximate solution were to be "okay?" It turns out that I can get an approximate solution as follows:

$$A\vec{x} = \vec{b} \text{ becomes instead } A^T A\vec{x} = A^T\vec{b}$$

where A^T is the transpose of A. Note that while A was a 5×2 matrix, since A^T is a 2×5 matrix, we have that $A^T A$ is a 2×2 matrix. Likewise, $A^T\vec{b}$ is a 2-dimensional vector. If you are unfamiliar with the transpose operation, note that it is just interchanging the rows and the columns of the matrix. This means that the old rows are the new columns, and the old columns are the new rows. In our case, this means that

$$A^T = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Therefore, I type into Sage the following code:

```
A = matrix(5, 2, [0, 1, 1, 1, 2, 1, 3, 1, 4, 1] )
print A
print

b = matrix(5, 1, [7.1, 5.2, 2.9, 1.05, -0.9] )
print b
print

M = A.transpose() * A
d = A.transpose() * b

M \ d
```

The output indicates that I have $m = -403/200$ or $m = -2.015$ and $b = 71/10 = 7.1$ as my solution. Do these numbers look familiar? It turns out that this is precisely and exactly the linear regression problem that we had on Page 157. Whenever you are computing a linear regression (i.e., a best-fit line) on n data points, then you are solving n linear equations in 2 unknowns, using this funky method, called "Least Squares." The method was invented

by Carl Friedrich Gauss (1777–1855), or by Adrien-Marie Legendre (1752–1833), depending on which history of mathematics book you consult.

If we think of the data points as

$$(x_1, y_1),\ (x_2, y_2),\ \ldots, (x_5, y_5)$$

then we could define the error in the ith datapoint as

$$e_i = y_i - \hat{y}_i = y_i - mx_i - b$$

where $\hat{y}_i = mx_i + b$ is the estimated value of y_i, predicted by the best-fit line. This least squares procedure guarantees three amazing results:

- The sum of the e_is is zero.
- The length of $\vec{e_i}$, or equivalently, the sum: $e_1^2 + e_2^2 + \cdots + e_n^2$ is as small as possible. Note that $||\vec{e_i}||^2$ is equal to that sum.
- The point $(\overline{x}, \overline{y})$ is guaranteed to be on the line $y = mx + b$. Here $\overline{x}$ is the average value of the x_is, and $\overline{y}$ is the average value of the y_is.

Last but not least, the entries of A^TA and A^Tb have meaning. They are

$$A^TA = \begin{bmatrix} 30 & 10 \\ 10 & 5 \end{bmatrix} \text{ as well as } A^T\vec{b} = \begin{bmatrix} 10.55 \\ 15.35 \end{bmatrix}$$

As you can see, the 30 is the sum of the squares of the x_is. The two 10s are the sums of the x_is. The 5 is the number of data points. The 15.35 is the sum of the y_is. Last but not least, the 10.55 is the sum:

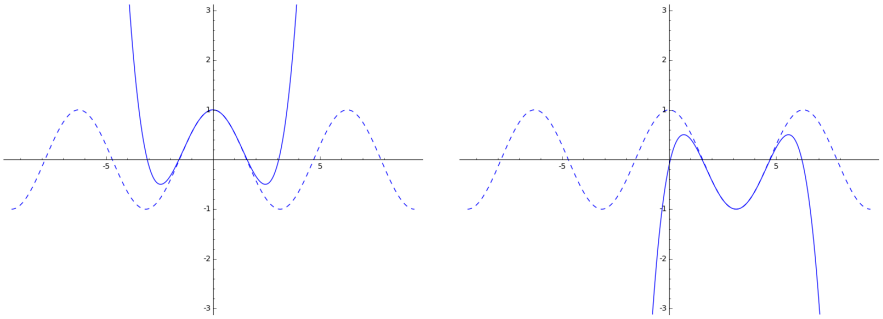
$$x_1y_1 + x_2y_2 + \cdots + x_5y_5$$

It should be noted that there are many times when converting $A\vec{x} = \vec{b}$ into $A^TA\vec{x} = A^T\vec{b}$ makes sense. Computing linear regressions is just one application. The smaller system of linear equations produced by this trick is referred to as "the normal equations."

4.17. Computing Taylor or MacLaurin Polynomials

Taylor polynomials make excellent approximations to functions, and naturally they come up in many applications of calculus to the sciences—most especially in physics. Unfortunately, computing a Taylor polynomial by hand can be a major undertaking. Often one has to compute 8 terms in order to get a mere 4 non-zero terms. This, in turn, requires taking a 7th derivative. Luckily, Sage can do this for you.

First, it is worthwhile to remind ourselves what the definition of a Taylor polynomial is. Recall that a line $\ell(x) = mx + b$ is said to be the tangent line of some function $f(x)$ at the point $x = x_0$ if the tangent line "touches" $f(x)$ at x_0, and has "the same slope as" $f(x)$ at x_0. In precise terms, that means the line has $f'(x_0)$ as its slope, and $f(x_0) = \ell(x_0)$. We could define a parabola $p(x)$ to be the tangent parabola at x_0 similarly. It would have to have $p(x_0) = f(x_0)$, and $p'(x_0) = f'(x_0)$, as well as $p''(x_0) = f''(x_0)$.



Taylor Polynomial about $x = 0$ Taylor Polynomial about $x = \pi$

FIGURE 3. Two Examples of Taylor Polynomials of $f(x) = \cos x$.

Analogous to the tangent line and tangent parabola, the nth degree Taylor polynomial of $f(x)$ at x_0 is the unique nth degree polynomial $p(x)$ such that $p(x_0) = f(x_0)$, and $p'(x_0) = f'(x_0)$, as well as $p''(x_0) = f''(x_0)$, and so forth, until the nth derivative. The definition does not restrict the $n+1$th or higher derivatives—they can be different, or they can be the same.

Oddly, some math faculty insist that if $x_0 = 0$, that the polynomial be called a MacLaurin polynomial, whereas if $x_0 \neq 0$, it is to be called a Taylor polynomial. This rule is extremely strange. The mathematicians involved are Brook Taylor (1685–1731) and Colin Maclaurin (1698–1746), as well as James Gregory (1638–1675). Sadly, each had an unusually short life, even by the standards of the their own times.

4.17.1. Examples of Taylor Polynomials

First, we will compute the 5th degree taylor polynomial of $f(x) = \cos x$ at the point $x = 0$. The code below also will plot the cosine function (as a dotted curve), and the Taylor polynomial (as a solid curve).

```
f(x) = cos(x)
p(x) = taylor( f(x), x, 0, 5 )

print "Polynomial:"
print p(x)

P1= plot( p(x), -3*pi, 3*pi, ymax=3, ymin=-3 )
P2= plot( f(x), -3*pi, 3*pi, linestyle='--' )
show(P1 + P2)
```

The output produced is the polynomial itself:

```
 1/24*x^4 - 1/2*x^2 + 1
```

as well as the plot given on the left in Figure 3.

Meanwhile, we can change the `0` in the second line of our code to be `pi`. Then we get the 5th degree Taylor polynomial of $f(x)$ at the point $x = \pi$. That's the plot above, on the right.

In summary, the `taylor` command expects four inputs: first, the function; second, the variable; third, the point about which the approximation is being made; fourth, the degree of the approximation required. Here, we asked for the 5th degree and got the 4th degree. That's because the fifth-degree coefficient turns out to be zero. Since we don't normally write $0x^5$, that fifth-degree term is missing entirely.

By the way, as this is a four-parameter function, it happens that I almost always get confused on the order of the four parameters. "Real programmers" always check the documentation. No one actually remembers fine details like the particular order of the inputs. Even if you are confident that you remember which is which, it is better to check anyway (using the techniques of Section 1.10 on Page 47) to avoid the possibility of being mistaken.

4.17.2. An Application: Understanding How g Changes

In *Physics I*, when we work with problems that take place on Earth (and near its surface) we just set $g = 9.82$ m/s^2 and then we can compute the force of gravity (i.e. the weight) of any mass via $w = mg$, or $w = -mg$, depending on what sign conventions you prefer. When we work on problems involving satellites, nontrivially far from the earth's surface, we learn Newton's formula for the force of gravity:

$$F = \frac{Gm_e m_s}{r^2}$$

where G is the universal constant of gravitation, m_e is the mass of the earth, m_s is the mass of the object (usually a satellite), and r is the distance from the object to the center of the earth.

That's all rather excellent—but what do we do for distances that are in between? If we are 1 meter above the earth's surface, then we know to use $g = 9.82$ m/s^2. If we are 10^6 meters above, then for sure, we should use Newton's formula. However, it would be nice to know how far off we are if we use $g = 9.82$ m/s^2 in a problem, for example, about a jet-liner, or about Mount Everest. We will know explore this issue by making a Taylor polynomial representation of Newton's formula. The sizes of the coefficients will tell us much information.

If we set $F = w = m_s g$, then we are finding out what value "little g" takes at this particular r value. See the following calculation:

$$\begin{aligned} F &= \frac{G m_e m_s}{r^2} \\ m_s g &= \frac{G m_e m_s}{r^2} \\ g &= \frac{G m_e}{r^2} \\ g &= \frac{G m_e}{(r_e + x)^2} \end{aligned}$$

As you can see, the m_s on the left cancels out the m_s on the right. This should not be surprising, because just as $g = 9.82$ m/s^2 for all objects at the earth's surface, regardless of their mass, so should the local value of g far from the earth's surface not depend on m_s. Last, you will see that we substituted $r = r_e + x$ at the last step. The r_e represents the radius of the earth, and the x is the distance from the earth's surface to the object—normally called the object's altitude. The reason that we do this is that it is easy to measure the altitude of an object with an altimeter—but it is rather difficult to slice the entire earth in half simply for the benefit of placing one end of an enormous ruler at its center.

Now we will take this function

$$g(x) = \frac{G m_e}{(r_e + x)^2}$$

and compute a Taylor polynomial at $x = 0$, which is the earth's surface. Should we use a first-degree polynomial? A second-degree? Maybe a third-degree? At this point in the problem, it is not obvious. My strategy (from my engineering days) is to make a high-degree approximation and then throw out most of the coefficients. The magnitudes of the coefficients will tell me which ones to throw out. Therefore, I'm going to ask Sage to produce a sixth-degree approximation. The code is below:

```
G = 6.67384*10^(-11)
Me = 5.97219*10^24
re = 6.371*10^6
g(x) = G*Me/(re + x)^2

p(x) = taylor( g(x), x, 0, 6 )

print "Polynomial:"
print p(x)

print "Just to Confirm:"
print g(0)
print p(0)
```

That code produces the following output:

```
Polynomial:
(1.02788991701e-39)*x^6 - (5.61315999537e-33)*x^5 + (2.98012019421e-26)*x^4
     - (1.51890766058e-19)*x^3 + (7.25772052919e-13)*x^2
     - (3.08259583276e-06)*x + 9.81960902527
Just to Confirm:
9.81960902526829
9.81960902527
```

The lines "just to confirm" are there to remind us that at $x = 0$, we are at the earth's surface and we expect to use $g = 9.82$ m/s^2. That seems to match. As for the polynomial itself, the coefficients of the third, fourth, fifth, and sixth degree terms are exceptionally small—and therefore we can delete them. For most problems we could delete the quadratic coefficient also, but if perhaps $x = 2 \times 10^6$ m^2 then $x^2 = 4 \times 10^{12}$ m^2, and thus the coefficient of 7.25×10^{-13} seems to be worth keeping in that case.

The constant coefficient is reassuring, reminding us that $g = 9.82$ m/s^2 at the earth's surface. Notice that the number 9.82 m/s^2 does not appear anywhere in our Sage program—it is not an input—it was computed mathematically. The linear coefficient is the most important one of all in this case. Because it is -3.08259×10^{-6} we know that if $x \approx 1000$ m, then we have nothing to worry about. We can use 9.82 m/s^2 with impunity. On the other hand, if $x \approx 10^6$ m, then 9.82 m/s^2 is a very poor approximation. Moreover, every time we go up 1 km from the earth's surface, g is reduced by -0.003 m/s^2. This means that even at 10^4 m altitude, we might want to consider using Newton's formula instead of 9.82 m/s^2.

4.18. Minimizations and Lagrange Multipliers

Finding the minimum or maximum of a single-variable function is a common task in *Calculus I*. For a multivariable function, the problem is a bit harder. Sage has several time-saving commands built in for multivariable optimization.

There are two categories: constrained and unconstrained optimization.

4.18.1. Unconstrained Optimization

Let us suppose that you must find the minima of the function

$$f(x, y, z) = 120(y - x^2 + 1)^2 + (2 - x)^2 + 110(z - y^2)^2 + 150(3 - y)^2$$

One option, perhaps the most classic, is to take the gradient, ∇f, and set it equal to the zero vector. The vector equation $\nabla f = \vec{0}$ has three unknowns and three equations, but it is nonlinear. Since the system is small, it might be solvable using resultants but that's a lot of advanced work.

Often, a numerical solution will suffice, but this requires an initial guess or initial condition. Perhaps your initial guess is $x = 0.1$, $y = 0.3$, and $z = 0.4$. Then you would type the following.

```
var('y z')
f(x, y, z)=120*(y-x^2+1)^2 + (2-x)^2 + 110*(z - y^2)^2 + 150*(3-y)^2
minimize(f, [0.1, 0.3, 0.4])
```

Then we receive the output

```
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 14
         Function evaluations: 22
         Gradient evaluations: 22

(2.0000000019, 3.00000000582, 9.00000003228)
```

We can have high confidence that a good minimum is $x = 2$, $y = 3$, and $z = 9$. As you can see, 14 iterations of gradient descent[3] were performed. However, we might be curious to see how much our answer depends on initial conditions. Let's try a bad guess, perhaps $x = 10$, $y = 30$, and $z = 40$. We will get the same answer, but it will take 50 iterations instead.

In the case of a function with many local minima, it is not obvious which minima will be found. The domain of the function is divided into "basins of attraction." Those are sets of points in the domain that all will converge to the same local minimum. However, the basins themselves can be very complicated, and in extreme cases, they can be fractal. The same is true for maximizations.

Convex functions, however, have only one local minimum or one local maximum, which is therefore also global. Moreover, they cannot have both a global minimum and a global maximum, only one of those two. In single-variable calculus, those are functions where either $f''(x)$ is always positive (a unique minimum), or $f''(x)$ is always negative (a unique maximum). However, it isn't too clear what the equivalent notion would be for multivariate functions.

For a multivariate function such as $f(x, y, z)$ to have a unique maximum or minimum, it turns out that the Hessian matrix H, see Page 204, must have either only positive eigenvalues (unique minimum), or only negative eigenvalues (unique maximum). Some readers might not know about Hessian matrices. For functions of only two variables, there is a really awesome shortcut: $f(x, y, z)$ has a unique minimum (or maximum) if

$$\left(\frac{\partial^2 f}{\partial x \partial x}\right)\left(\frac{\partial^2 f}{\partial y \partial y}\right) - \left(\frac{\partial^2 f}{\partial x \partial y}\right)^2 > 0$$

but note that this is not an "if and only if" relationship.

[3]Gradient descent is one of my favorite algorithms in all of mathematics, but I do not really have the space to describe it here. Suffice it to say that it was discovered by Augustin-Louis Cauchy (1798–1857), who is better known for giving us the modern notion of convergence and divergence of infinite series, and other results in *Real Analysis*.

These "very well behaved" problems (that is to say, convex problems) come up in mathematical economics relatively often. They have the nice property that gradient descent will always succeed, and always converge to the unique optimum point.

4.18.2. Constrained Optimization by Lagrange Multipliers

In general, if I have a function $f(x_1, x_2, \ldots, x_n)$ and I want to minimize it subject to the constraint function $g_i(x_1, x_2, \ldots, x_n) = 0$ for several constraints g_1, g_2, ..., g_m, then I should define a new function

$$\mathcal{F}(x_1, x_2, \ldots, x_n, \lambda_1, \lambda_2, \ldots, \lambda_m) = f(\vec{x}) + \lambda_1 g_1(\vec{x}) + \lambda_2 g_2(\vec{x}) + \cdots + \lambda_m g_m(\vec{x})$$

where the new variables, $\lambda_1, \lambda_2, \ldots, \lambda_m$ are called "Lagragne[4] Multipliers."

The unconstrained, genuine minimum of $\mathcal{F}$ will be the minimum of f on the set of all points which satisfy all m constraints simultaneously—assuming that set is not empty. We also need the technical assumption that $\nabla g_i \neq \vec{0}$ for all points where $g_i(\vec{x}) = 0$, for each of the g_is. This technical assumption tends to be true in application problems.

4.18.3. A Lagrange Multipliers Example in Sage

The following example is take from James Stewart's *Calculus*, 6th edition, where it is Example 4 from Section 15.8, "Lagrange Multipliers." Find the point on the sphere $x^2 + y^2 + z^2 = 4$ that is closest to the point $(3, 1, -1)$. We could try to minimize

$$D = \sqrt{(x-3)^2 + (y-1)^2 + (z+1)^2}$$

but it makes more sense instead to minimize

$$D^2 = (x-3)^2 + (y-1)^2 + (z+1)^2$$

which is legal because D is never negative. Because D is never negative, the task of minimizing D and minimizing D^2 are exactly the same act.

Thus we wish to minimize

$$f(x, y, z) = (x-3)^2 + (y-1)^2 + (z+1)^2$$

subject to

$$g(x, y, z) = x^2 + y^2 + z^2 - 4 = 0$$

which means we must minimize instead

$$\mathcal{F}(x, y, z, \lambda) = (x-3)^2 + (y-1)^2 + (z+1)^2 + \lambda(x^2 + y^2 + z^2 - 4)$$

so naturally we should find $\nabla\mathcal{F}$ and set it equal to $\vec{0}$.

[4]Named for Joseph-Louis Lagrange (1736–1813).

We begin by finding the four partial derivatives

$$\begin{aligned}\partial\mathcal{F}/\partial x &= 2(x-3)+\lambda(2)(x)\\ \partial\mathcal{F}/\partial y &= 2(y-1)+\lambda(2)(y)\\ \partial\mathcal{F}/\partial z &= 2(z+1)+\lambda(2)(z)\\ \partial\mathcal{F}/\partial \lambda &= x^2+y^2+z^2-4\end{aligned}$$

Resulting in four quadratic equations in four variables:

$$\begin{aligned}2(x-3)+2x\lambda=0\\ 2(y-1)+2y\lambda=0\\ 2(z+1)+2z\lambda=0\\ x^2+y^2+z^2-4=0\end{aligned}$$

We are simultaneously disappointed (because the equations are quadratic) and delighted (because the last equation clearly guarantees that any solution is on the sphere in question). After much algebra, we get two solutions

$$x=\frac{6}{\sqrt{11}},\ y=\frac{2}{\sqrt{11}},\ z=\frac{-2}{\sqrt{11}} \text{ as well as } x=\frac{-6}{\sqrt{11}},\ y=\frac{-2}{\sqrt{11}},\ z=\frac{2}{\sqrt{11}}$$

As you can see, the first solution is the point on the sphere *closest* to $(3,1,-1)$ and the second solution is the point on the sphere *farthest* from $(3,1,-1)$.

The Example in Sage

To do this in Sage, we'd have to compute $\mathcal{F}(x,y,z,\lambda)$ ourselves. We type

```
var("y z L")
F(x, y, z, L)=(x-3)^2 + (y-1)^2 + (z+1)^2 + L*(x^2 + y^2 + z^2 - 4)
minimize(F, [0,0,0,0] )
```

The reply from Sage is below:

```
Warning:
      Desired error not necessarily achieved due to precision loss.
      Current function value: 10.480396
      Iterations: 1
      Function evaluations: 28
      Gradient evaluations: 16

(1.9173321311, 0.639110710366, -0.639110710366, 1.27822142073)
```

This represents Sage failing. We have the warning, and also the "current function value" should be zero. I tried many different initial conditions, over and over. This was most frustrating. Rest assured that Sage's `minimize` command usually works better. However, because it happened to me, it can happen to you, so I'm going to share what you should do to get around this situation.

When Sage's Minimize Command Won't Minimize

When a mathematician hopes to minimize a multivariate function, a very common strategy is to compute the gradient, and set it equal to zero. Therefore, let's start by asking Sage to compute the gradient for us. The code below does that

```
var("y z L")
F(x, y, z, L)=(x-3)^2 + (y-1)^2 + (z+1)^2 + L*(x^2 + y^2 + z^2 - 4)
derivative(F)
```

Sage responds with

```
(x, y, z, L) |-->(2*L*x + 2*x - 6, 2*L*y + 2*y - 2, 2*L*z + 2*z + 2,
          x^2 + y^2 + z^2 - 4)
```

Using cut-and-paste, either in a different browser window running the Sage Cell Server, or alternatively in SageMathCloud, we can assemble the following equations, and solve them with the `solve` command.

```
eqn1= 2*L*x + 2*x - 6 == 0
eqn2= 2*L*y + 2*y - 2 == 0
eqn3= 2*L*z + 2*z + 2 == 0
eqn4= x^2 + y^2 + z^2 - 4 == 0

solve( [eqn1, eqn2, eqn3, eqn4],  [x, y, z, L] )
```

As you can see, Sage produces the exact minimum and maximum in this case:

```
[ [ x == -6/11*sqrt(11), y == -2/11*sqrt(11), z == 2/11*sqrt(11),
L == -1/2*sqrt(11)-1], [ x == 6/11*sqrt(11), y == 2/11*sqrt(11),
z == -2/11*sqrt(11), L == 1/2*sqrt(11)-1 ] ]
```

Furthermore, this solution has the advantage of being exact, and not being a mere numerical approximation. What is the moral of this story? If symbolic methods will work, don't bother with numerical methods. Numerical methods are sensitive to initial conditions, prone to rounding error, and often you have to carefully select parameters. Algebraic methods, when possible, circumvent all these flaws. On the other hand, algebraic methods are often not applicable—such as when finding the roots of polynomial equations of degree five and higher.

4.18.4. Some Applied Problems

There are numerous examples of Lagrange Multipliers, particularly in mathematical economics. I would first recommend Shapoor Vali's *Principles of Mathematical Economics*, published by Atlantis Press in 2013. In particular, Lagrange Multipliers appear in Chapter 10.4.2, "Production Function, Least Cost Combination of Resources, and Profit Maximizing Level of Output."

I would also recommend *Mathematics for Finance: An Introduction to Financial Engineering*, by Marek Capinski and Tomasz Zastawniak, published in 2003 by Springer under the Springer Undergraduate Mathematics Series. Sections 5.1, 5.2, and 5.3 introduce Markowitz's Optimal Portfolio

Strategy, essentially choosing stocks which are contra-variant to diminish investor risk. That strategy is basically one enormous Lagrange Multipliers problem.

4.19. Infinite Sums and Series

Let's start with a very simple sum:

$$-4+-1+2+5+8+11+14+17+20+23+26+29+32+35+\cdots+131+134+137$$

First, we identify that it is an arithmetic progression, because the difference between adjacent terms is exactly three in all cases. Further examination tells us that this progression can be thought of as $3k-7$ for $k \in \{1,2,3,\ldots,48\}$ or $3k-4$ for $k \in \{0,2,3,\ldots,47\}$. The former notation makes more sense to me.

To represent this in Sage, we should type

```
var("k")
sum( 3*k-7, k, 1, 48)
```

and we get the correct answer, namely 3192.

We might want a more general formula for

$$1+2+3+4+\cdots+(n-2)+(n-1)+n=\sum_{k=1}^{k=n} k$$

in which case we should type

```
var("k m")
print sum(k, k, 1, m)
print factor( sum(k, k, 1, m) )
```

We receive back the following response:

```
1/2*m^2 + 1/2*m
1/2*(m + 1)*m
```

One of the great open problems in mathematics is the Riemann[5] Hypothesis and it is related to the Riemann Zeta function. A sum of the form

$$\frac{1}{1}+\frac{1}{2^4}+\frac{1}{3^4}+\frac{1}{4^4}+\frac{1}{5^4}+\frac{1}{6^4}+\cdots=\sum_{k=1}^{k=\infty}\frac{1}{k^4}=\zeta(4)$$

is an example of evaluating the Reimann Zeta function at $s=4$. In general,

$$\frac{1}{1}+\frac{1}{2^s}+\frac{1}{3^s}+\frac{1}{4^s}+\frac{1}{5^s}+\frac{1}{6^s}+\cdots=\sum_{k=1}^{k=\infty}\frac{1}{k^s}=\zeta(s)$$

where s can be any complex number, except the negative integers. In Sage, we can evaluate the $s=4$ case with

[5]Named for Georg Friedrich Bernhard Riemann (1826–1866). He is the Riemann for Riemann Geometry (the curvature of space), the Riemann Hypothesis, and he also contributed a lot to the study of prime numbers.

```
sum(1/k^4, k, 1, oo)
```

We get the answer: $\pi^4/90$. The two consecutive letters "o" are the Sage symbol for infinity in this case, namely `oo`. If you squint and cross your eyes slightly, the `oo` looks like ∞.

4.19.1. Verifying Summation Identities

If you've worked with combinations, you might know the following identities:

$$\begin{aligned}\binom{n}{0}+\binom{n}{1}+\binom{n}{2}+\cdots+\binom{n}{n-2}+\binom{n}{n-1}+\binom{n}{n} &= 2^n\\ \binom{n}{0}-\binom{n}{1}+\binom{n}{2}-\binom{n}{3}+\binom{n}{4}-\binom{n}{5}+\binom{n}{6}-\cdots\pm\binom{n}{n} &= 0\end{aligned}$$

We can verify this with the following Sage code:

```
var("k m")
print sum(binomial(m,k), k, 0, m)
print sum((-1)^k*binomial(m,k), k, 0, m)
```

The response reveals that Sage knows these identities.

```
2^m
0
```

Another fun trick is that the sum of the first n cubes is equal to the square of the sum of the first n integers. In other words:

$$1^3+2^3+3^3+4^3+\cdots+(n-1)^3+n^3=(1+2+3+4+\cdots+n)^2$$

At first, I tried to verify this with the following code

```
var("k m")
sum(k, k, 1, m)^2 - sum(k^3, k, 1, m)
```

yet I received back the following answer:

```
 -1/4*m^4 - 1/2*m^3 + 1/4*(m^2 + m)^2 - 1/4*m^2
```

Then I tried the code below, and got back a more satisfying "`0`."

```
var("k m")
f(m) = sum(k, k, 1, m)^2 - sum(k^3, k, 1, m)
f(m).expand()
```

4.19.2. The Geometric Series

If we wanted to study a particular geometric progression

$$25+20+16+\frac{64}{5}+\frac{256}{25}+\frac{1024}{125}+\frac{4096}{625}+\cdots$$

we would start by dividing each element by the one before it, to discover that the common ratio is $c_r = 4/5$. Perhaps we could consider the general

form of a geometric progression:

$$a + a(c_r) + a(c_r)^2 + a(c_r)^3 + a(c_r)^4 + a(c_r)^5 + \cdots + a(c_r)^m = \sum_{k=0}^{k=m} a(c_r)^k$$

where a denotes the first element, and c_r the "common ratio."

We can compute this sum in Sage, using the code below:

```
var("a c_r k m")
sum( a*(c_r)^k, k, 0, m)
```

We get the formula below, which is correct.

```
(a*c_r^(m + 1) - a)/(c_r - 1)
```

For the infinite sum, we would just have k run from 0 to ∞ instead of from 0 to m. However, Sage makes an objection when we type the following code:

```
var("a c_r k m")
sum( a*(c_r)^k, k, 0, oo)
```

The answer we receive back is a huge error message, followed by:

```
 ValueError: Computation failed since Maxima requested additional
constraints; using the 'assume' command before summation *may* help
(example of legal syntax is 'assume(abs(c_r)-1>0)', see `assume?`
for more details)
Is abs(c_r)-1 positive, negative or zero?
```

The c_r denotes the common ratio, and from *Calculus II* we know that it should be the case that $-1 < c_r < 1$. Therefore, we can easily make the assumption that $|c_r| < 1$. We insert the assumption line as the new, middle line of our code:

```
var("a c_r k m")
assume( abs(c_r) < 1 )
sum( a*(c_r)^k, k, 0, oo)
```

With this added line, answer we receive back is the conventional answer.

```
 -a/(c_r - 1)
```

Here, we can see that Sage was very wise to force us to declare the assumption that $-1 < c_r < 1$, because the geometric progression will otherwise[6] diverge. By the way, the `assume` command comes up in integration, also. See Page 60 and Page 202 for examples of that.

4.19.3. Using Sage to Guide a Summation Proof

Suppose you are asked to compute what

$$\sum_{j=0}^{j=\infty} \frac{x^{2j+1}}{j!} = x + \frac{x^3}{1!} + \frac{x^5}{2!} + \frac{x^7}{3!} + \frac{x^9}{4!} + \frac{x^{11}}{5!} + \cdots$$

[6]With the notable but extremely silly exception of $a = 0$, where the series converges to 0 regardless of the choice of c_r.

converges to, and then give a proof of that. We will ignore "radius of convergence" issues for now, which actually do not come up in this problem.

First, let's ask Sage what that sum actually turns out to be, using the following code:

```
var("j")
sum( x^(2*j+1)/factorial(j), j, 0, oo)
```

The response is

```
x*e^(x^2)
```

which will be our objective in our proof.

Let's say that you are pretty sure that

$$e^y = 1 + \frac{y}{1} + \frac{y^2}{2!} + \frac{y^3}{3!} + \frac{y^4}{4!} + \frac{y^5}{5!} + \frac{y^6}{6!} + \cdots$$

but not entirely sure. You can verify this with

```
var("k y")
sum( y^k/factorial(k), k, 0, oo)
```

As you can see, Sage has helped us twice: first to give us a target for our proof, and second to help us confirm an important basic formula which we might or might not have misremembered. The rest is pretty straight forward, after we substitute $y = x^2$ into the definition of e^y. We have the following "proof," but of course a student would have to add some words to this.

$$\begin{aligned}
e^y &= 1 + \frac{y}{1} + \frac{y^2}{2!} + \frac{y^3}{3!} + \frac{y^4}{4!} + \frac{y^5}{5!} + \frac{y^6}{6!} + \cdots \\
e^{x^2} &= 1 + \frac{x^2}{1} + \frac{x^4}{2!} + \frac{x^6}{3!} + \frac{x^8}{4!} + \frac{x^{10}}{5!} + \frac{x^{12}}{6!} + \cdots \\
xe^{x^2} &= x + \frac{x^3}{1} + \frac{x^5}{2!} + \frac{x^7}{3!} + \frac{x^9}{4!} + \frac{x^{11}}{5!} + \frac{x^{13}}{6!} + \cdots \\
xe^{x^2} &= \sum_{j=0}^{j=\infty} \frac{x^{2j+1}}{j!}
\end{aligned}$$

Philosophical Point

Of course, the point of all this is that Sage is to be a set of training wheels for the student's learning of this famously hard topic: infinite series. Sage can help students get over the initial befuddlement and confusion when being first exposed to this material. The goal is that, at least after a while, the student should consult Sage less frequently, until the problems can be solved without Sage's help at all. Only then can a rigorous *Calculus II* exam be a success.

4.20. Continued Fractions in Sage

This tiny section assumes that you know something already about continued fractions. Let's suppose that you wanted to know the continued fraction expansion of $\sqrt{11}$. Then you should type

```
continued_fraction( sqrt(11) )
```

returns the output

```
[3, 3, 6, 3, 6, 3, 6, 3, 6, 3, 6, 3]
```

which is Sage's way of telling you that

$$\sqrt{11} = 3 + \cfrac{1}{3 + \cfrac{1}{6 + \cfrac{1}{3 + \cfrac{1}{6 + \cfrac{1}{3 + \cfrac{1}{6 + \cfrac{1}{3 + \cfrac{1}{6 + \cfrac{1}{3 + \cfrac{1}{6 + \cdots}}}}}}}}}}$$

However, in applied mathematics, we usually want to use continued fractions to obtain some close but compact rational approximations for irrational numbers. There are other uses of continued fractions, but this particular use is so frequent that a shortcut has been made. You can type

```
c = continued_fraction( sqrt(11) )
c.convergents()
```

which results in the following output:

```
[3,
 10/3,
 63/19,
 199/60,
 1257/379,
 3970/1197,
 25077/7561,
 79201/23880,
 500283/150841,
 1580050/476403,
 9980583/3009259,
 31521799/9504180]
```

That output is a sequence of rational numbers, each getting closer and closer to $\sqrt{11}$. Continued fraction approximations are extremely efficient. For example, 1257/379 has a relative error of $-6.32891\cdots \times 10^{-7}$, which is pretty impressive considering that we only have a 3-digit denominator and a 4-digit numerator. If we change the above code to get the continued fraction approximations (what Sage calls "convergents") of π, then we see

```
[3,
 22/7,
 333/106,
 355/113,
```

```
103993/33102,
104348/33215,
208341/66317,
312689/99532,
833719/265381,
1146408/364913,
4272943/1360120,
5419351/1725033,
80143857/25510582]
```

The first four of those contain three very famous approximations. Many of us were taught $\pi \approx 22/7$ in high school. The approximation 355/113 was discussed extensively on Page 32. The approximation 3 is said to have been used by several ancient civilizations, including the Bible. See the article "The Chronology of Pi" by Herman C. Schepler, in *Mathematical Magazine*, Vol 23, No. 3, 1950. An interesting rebuttal can be found at

`http://www.purplemath.com/modules/bibleval.htm`

4.21. Systems of Inequalities and Linear Programming

An unbelievable diversity of problems can be solved with concept of linear programming. These include routing shipments, scheduling, ingredient mixes for highly processed foods, portfolio balancing, inventory planning, advertising allocations, and a horde of problems in manufacturing, not to mention finding Nash Equilibria in zero-sum games.

We don't have the space to go into all of those (in fact, good coverage of that material often requires an entire book much larger than this one). However, we will see how to use Sage to solve linear programs that are presented in symbolic form.

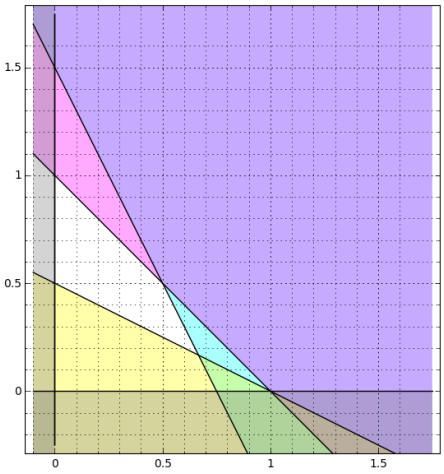
4.21.1. A Simple Example

Here is a simple example:

$$\begin{aligned} Maximize & & 2x + y \\ subject & to: & \\ 1.5 - 2x & \geq & y \\ 1 - x & \geq & y \\ 1 - x & \leq & 2y \\ x & \geq & 0 \\ y & \geq & 0 \end{aligned}$$

By the way, if you're curious to see what that example looks like graphically, it is given below. The shaded regions are those whose points do not satisfy all the inequalities (i.e. infeasible points), while the unshaded

region is the one whose points do satisfy all the inequalities (i.e. feasible points). Incidentally, code for making graphs of this type will be included in the electronic-only online appendix to this book "Plotting in Color, in 3D, and Animations," available on my webpage `www.gregorybard.com` for downloading.



Furthermore, this system of inequalities is the subject of an interactive[7] webpage, suitable for students even in extremely remedial classes. In any case, to solve the linear program, we would type the following:

```
LP = MixedIntegerLinearProgram( maximization = True )

x = LP.new_variable()

LP.add_constraint( 1.5 - 2*x[1] >= x[2] )
LP.add_constraint( 1 - x[1] >= x[2] )
LP.add_constraint( 1 - x[1] <= 2*x[2] )

LP.add_constraint( x[1] >= 0 )
LP.add_constraint( x[2] >= 0 )

LP.set_objective( 5*x[1] + 4*x[2] )

print LP.solve()

print LP.get_values( x )
```

We will now go line-by-line through that code.

- The first line is the least intuitive. The name `LP` gives a name to this linear program, and it can be whatever name you'd like to give it. The long word `MixedIntegerLinearProgram` must always appear, in that capitalization, without spaces. Then you will indicate

[7] `http://www.gregorybard.com/interacts/feasible_region.html`

whether the problem is a maximization, as we've done in this case, or alternatively you can indicate if the problem is a minimization by saying `= False`.

- Then we declare our variable family `x`. This actually declares an infinite set of variables, `x[0]`, `x[1]`, `x[2]`, `x[3]`, ... As you can see in this example, we only need `x[1]` and `x[2]`. You can even use indices non-consecutively. We could have used the indices `x[420]` and `x[88]` instead of `x[1]` and `x[2]` had we reason to do so.
- After this come the problem's constraints. As you can see, these are fairly straight-forward. You must type `>=` for $\geq$ and `<=` for $\leq$. Don't forget to put an asterisk between the coefficients and the variables.
- Next we have to provide the minor constraints—that the variables `x[1]` and `x[2]` must never be negative.
- The line after that declares the objective function. Again, this is straight-forward Sage notation.
- The `print` statement will output the optimal value (in this case, the maximum value) of the objective function.
- The line after that might seem puzzling at first. This line will print the values of the variables that achieve this optimum. It might appear strange that a separate line, after `LP.solve()`, is required. However, when we talk about "multiple variable names" below, you'll see why this is there.

The output of that code should be the following:

```
4.5
{1: 0.5, 2: 0.5}
```

However, if you try this example shortly after this book is published, you might receive instead what is below.

```
4.5
{1: 0.5, 2: 0.5}
/home/sageserver/sage/ipython/IPython/core/interactiveshell.py:2883:
DeprecationWarning: The default behaviour of new_variable() will
soon change! It will return 'real' variables instead of nonnegative
ones.
Please be explicit and call new_variable(nonnegative=True) instead.
See http://trac.sagemath.org/15521 for details.
  exec(code_obj, self.user_global_ns, self.user_ns)
```

Sage is going through a change right now. It has to do with the non-negativity of variables inside of linear programs. The code presented in this section will work both before the change and after the change. It would be tedious and confusing to describe the change here. However, this warning is displayed "for a while" before the change, so that users are aware that a change is coming.

4.21.2. Convenient Features in Practice

The following additional features can be found in Sage, relating to linear programming.

- If you have 20 or 30 variables, then repeatedly having the line

  ```
  LP.add_constraint( x[27] >= 0 )
  ```

  could be tedious to type. Instead, you can type

  ```
  x = new_variable(nonnegative=True)
  ```

  in place of the line `x = new_variable()`. Then all indices of `x` will be assumed to be greater than or equal to zero, with just one line of code. You do not have to write a separate inequality for each `x` variable.
- Interval constraints come up from time to time, and being able to use them in Sage can reduce the number of constraints by half in some cases. Some computer algebra systems require you to type $2 \leq 3x_4 + 7x_8 \leq 5$ as two separate constraints: $2 \leq 3x_4 + 7x_8$ and $3x_4 + 7x_8 \leq 5$. However, in Sage you can do this in one line:

  ```
  LP.add_constraint( 2 <= 3*x[4] + 7*x[8] <= 5 )
  ```
- Sometimes we wish to say $5x_1+6x_2 = 300$. Some computer algebra systems require you to do this with two constraints: $5x_1+6x_2 \leq 300$ and $5x_1 + 6x_2 \geq 300$. However, in Sage, this can be done in one line:

  ```
  LP.add_constraint( 5*x[1] + 6*x[2] == 300 )
  ```

  Note, that is two consecutive equal signs.
- There are some applications where it is enough to find a feasible solution, and there is no objective function. The following trick probably only matters in really huge linear programs, where the running time might be extremely long. Rather than setting the objective function to 0, to have a faster program, one should type

  ```
  LP.set_objective( None )
  ```
- If you type `LP.show()` then you get a human-readable summary of the linear program. This can be really useful for debugging.
- You can add and remove linear constraints as you go, before or after the solve command. There are certain cases where adding a constraint, solving again, adding a constraint, solving again, and so forth are very useful strategies. Each `LP.solve()` command knows about the constraints that are above it in the program (except those that have been removed), but not those that are given below it. These situations come up in "Goal Programming" and in the method of "Gomery Cuts."
- To remove a constraint you just type

  ```
  p.remove_constraint(17)
  ```

where 17 is the number of the constraint in the list shown when you type `LP.show()`. Remember, computer scientists count from 0, so the first constraint show is #0, followed by #1, and then #2. Accordingly, #17 is the 18th constraint shown.

- Imagine that I'm working on a shipping program with 10 factories and 300 dealerships. It might be nice for me to be able to denote the number of items from factory i sent to dealership j with s_{ij}. In Sage, this can be done. All you have to do is type

  ```
  s = new_variable( nonnegative=True, dim = 2 )
  ```

  and then you can use variables such as `s[2][3]`, or `s[1][5]` and so forth. Note that `s[5][7]` is not the same thing as `s[7][5]`. Naturally, you can also leave out the `nonnegative=True` if you wish to have negative quantities shipped from factories to dealerships.
- Sometimes having all the variables merely be named x is uninformative and hampers an efficient comprehension of the problem. For example, in the project described in Section 2.3 on Page 71, I had typed:

  ```
  d = myprogram.new_variable()
  t = myprogram.new_variable()
  g = myprogram.new_variable()
  m = myprogram.new_variable()
  ```

  This was done so that I could carry out the following variable plan. Let $d_1, d_2, \ldots, d_7$ be the amount shipped out of Duluth/Superior. Let $t_1, t_2, \ldots, t_7$ be the amount shipped out of Two Harbors. Let $g_1, g_2, \ldots, g_7$ be the amount shipped out of Green Bay. Let $m_1, m_2, \ldots, m_7$ be the amount shipped out of Minneapolis. To me, this is much easier than having x_1, x_2, x_3, x_4, ..., x_{26}, x_{27}, x_{28}.

  However, a consequence of this is that you need to ask for the variables separately. Despite the effort, this results in output that is much more human-readable than before.

  ```
  print "Duluth:"
  print myprogram.get_values(d)
  print "Two Harbors:"
  print myprogram.get_values(t)
  ```

  and so forth.

4.21.3. The Polyhedron of a Linear Program

You are probably familiar with the way of solving a linear program by drawing lines in the plane. If not, see the interactive webpage:

`http://www.gregorybard.com/interacts/feasible_region.html`

Just as two-variable problem divides the coordinate plan into (convex) polygonal regions, three-variable problems divide ordinary three-dimensional

space into (convex) polyhedra. Sage can graph these, and that is described in the electronic-only online appendix to this book "Plotting in Color, in 3D, and Animations," available on my webpage `www.gregorybard.com` for downloading.

4.21.4. Integer Linear Programs and Boolean Variables

The set of problems that can be modeled by linear programming becomes considerably larger when some variables are permitted to be designated as "integer-only." If only a few variables are so designated, then the problem is solved as several linear programs, using a method called "Gomery Cuts" or another method called "Branch and Bound." If many variables are so designated, then the problem might take a very long time to solve.

In any case, Sage has this feature built-in. To show that the variables `x[i]` can only be non-negative integers, whereas the variables `y[i]` can be non-negative real numbers, just type

```
x = LP.new_variable( nonnegative=True, integer=True )
y = LP.new_variable( nonnegative=True )
```

Furthermore, you can type

```
y = LP.new_variable( nonnegative=True, integer=False )
```

to really drive home the point to a human reader.

Boolean variables also come up from time to time. There are variables that can only be permitted to be 0 or 1, and nothing else. To show that all the variables `z[k]` are of this form, we type

```
z = LP.new_variable( binary=True )
```

4.21.5. Further Reading on Linear Programming

Like many topics, choosing what book to read on Linear Programming depends upon if you are looking for information about theory or practice.

- For the theory of how large linear programs are solved with the simplex method, I recommend *Linear Programming: An Introduction With Applications*, by Alan Sultan, self-published in the CreateSpace Independent Publishing Platform in 2011. This book is suitable for students who have not had much coursework in mathematics, as well as those who have.
- For the myriad of applications, a great (and remarkably understandable) source is *Optimization in Operations Research* by Ronald Rardin, published by Prentice Hall in 1997.

4.22. Differential Equations

I think for many of us, we were told at a young age that mathematics is the key to understanding the universe, or at least the universe of science.

This attribution has been made by many through the centuries, but its most famous[8] phrasing is by Galileo (1564–1642) in *The Assayer*,

> . . . in this grand book—I mean the universe—which stands continually open to our gaze, but it cannot be understood unless one first learns to comprehend the language and interpret the characters in which it is written. It is written in the language of mathematics, and its characters are triangles, circles, and other geometrical figures, without which it is humanly impossible to understand a single word of it; without these, one is wandering around in a dark labyrinth.

However, it is often not until the course *Differential Equations*, relatively late in the mathematical education of some students, that this promise is delivered. Suddenly, with the contents of that course, many of the hardest problems in physics, chemistry, ecology, and finance are rendered transparent and facile—to the point where they become mere homework problems.

Sadly, realistic problems from actual applications are often dirty—the coefficients have decimal points, the equations are not cleanly solvable (especially if you include air resistance), and some problems are just plain difficult. This where Sage can be of a genuine service to a student. The student can focus on modeling phenomena and correctly posing the problem—the tedium of numerical solutions or series solutions can be best delegated to the computer. By this division of labor between man-and-machine, considerably harder and more realistic problems can be tackled than by a single human working in isolation. This topic is the crown jewel of applied mathematics, and the author strongly encourages any student to explore far beyond the boundaries of the undergraduate course and penetrate deeply into advanced topics.

Luckily, an entire book has been written to teach the *Differential Equations* course using Sage. The book is *Introduction to Differential Equations Using Sage*, by David Joyner and Marshall Hampton, published by Johns Hopkins University Press in 2012. That book covers a wide array of topics related to differential equations, and therefore it would be redundant to reproduce all of that here.

Instead, we will show how to solve simple differential equations, then move on toward initial-value problems, and the construction of slope fields. Then I present a capstone problem which is modeling the trajectory of a torpedo, including the force of hydrodynamic drag. While Sage is extremely capable of numerical solutions (with Runge-Kutta methods), series solutions, boundary-value problems, systems of ordinary differential equations, separation of variables, and so forth—the reader is referred to the Joyner-Hanson book for those procedures.

[8]Here I use the standard translation, by Stillman Drake.

4.22.1. Some Easy Examples

If we wanted to solve the differential equation

$$\frac{dy}{dx} + y = 7$$

then we have to type the following:

```
y = function('y',x)
h = desolve( diff(y,x) + y == 7, y)

print "Unsimplified:"
print h
print "Simplified:"
print h.expand()
```

The first line is a bit new to us. Just as we learned to declare unknown variables with the `var` command back on Page 38, we must declare unknown functions also. The method of declaring an unknown function is with the `function` command, noted above. The only purpose of this is to tell Sage and Python that we don't know what $y(x)$ is going to turn out to be. In any case, we get the following response:

```
Unsimplified:
(c + 7*e^x)*e^(-x)
Simplified:
c*e^(-x) + 7
```

You can also change `diff(y,x)` into `diff(y,x,2)` to make the second derivative, rendering the differential equation

$$\frac{d^2y}{dx^2} + y = 7$$

which results in the response:

```
Unsimplified:
k2*cos(x) + k1*sin(x) + 7
Simplified:
k2*cos(x) + k1*sin(x) + 7
```

Now let's change the line with `h` to a new differential equation, representing

$$y\,\frac{dy}{dx} + \sin x = 0$$

using the code:

```
h = desolve(y*diff(y,x)+sin(x)==0, y)
```

We get a rather different sort of answer:

```
Unsimplified:
-1/2*y(x)^2 == c - cos(x)
Simplified:
-1/2*y(x)^2 == c - cos(x)
```

Here, $y(x)$ has been defined implicitly, not explicitly. However, we can utilize a bit of algebra and see what is really intended:

$$\begin{aligned}(-1/2)(y(x))^2 &= c - \cos x \\ (y(x))^2 &= 2\cos x - 2c \\ y(x) &= \pm\sqrt{2\cos x - 2c}\end{aligned}$$

The $\pm$ indicates that Sage could not have given us a single function as an answer. Because of the $\pm$, as written, $y(x)$ would fail the "vertical line test" and thus cannot be considered a function. In other words, for any function, choose any specific x in its domain, then there can only be one y-value for that x-value. Here, we would have two y-values, and that's not allowed. The better way to think of it is to imagine two solutions:

$$\begin{aligned}y_1(x) &= \sqrt{2\cos x - 2c} \\ y_2(x) &= -\sqrt{2\cos x - 2c}\end{aligned}$$

Actually, there's an infinite number of functions that satisfy the differential equation, because we can pick any value of c that we want. Let's say that we choose to include the minus sign (i.e by picking $y_2(x)$) and that we choose $c = 832$. How can we verify that this is indeed a solution?

We should type the following code:

```
f(x) = -sqrt(-2*(832-cos(x)))

print f(x)*diff( f(x), x) + sin(x)
```

We get back the response "`0`," which means that the original differential equation is indeed satisfied. I have the habit of always checking the solutions, because there are so many opportunities to mistype a function—perhaps by leaving out a minus sign or typing the wrong numeral. Since Sage will do all the work, it is a pity not to check.

4.22.2. An Initial-Value Problem

Now we're going to program Sage to solve this differential equation:

$$\frac{d^2y}{dx^2}\, y + y = x$$

subject to some initial-value constraints. If we try to solve the problem in the same style as what we did before, we would type

```
y = function('y', x)
myequation = diff(y, x, 2) + y == x
desolve(myequation, y )
```

That code produces the output:

```
k2*cos(x) + k1*sin(x) + x
```

That's easy enough to check by hand:

$$\begin{aligned} f(x) &= k_2(\cos x) + k_1(\sin x) + x \\ f'(x) &= -k_2(\sin x) + k_1(\cos x) + 1 \\ f''(x) &= -k_2(\cos x) - k_1(\sin x) \\ f''(x) + f(x) &= (-k_2(\cos x) - k_1(\sin x)) + (k_2(\cos x) + k_1(\sin x) + x) \\ &= (-k_2 + k_2)(\cos x) + (-k_1 + k_1)(\sin x) + x \\ &= x \end{aligned}$$

As it turns out, if we have an initial value problem, such as $y = 2$ at $x = 10$ and $dy/dx = 1$ at $x = 10$, then we will make only one tiny modification. We change the third line to

```
desolve(de, y, ics=[10,2,1])
```

where `ics` refers to "initial conditions." The `[10, 2, 1]` signifies that $y(0) = 10$, $y'(0) = 2$, and $y''(0) = 1$. Now we get the following answer:

```
x - 8*cos(10)*cos(x)/(cos(10)^2 + sin(10)^2) -
          8*sin(10)*sin(x)/(cos(10)^2 + sin(10)^2)
```

Of course, $\sin^2(10) + \cos^2(10) = 1$, so we should interpret this as

$$f(x) = x - (8\cos 10)(\cos x) - (8\sin 10)(\sin x)$$

If we want to check our work, we need a slightly longer snippet of code:

```
y = function('y', x)
myequation = diff(y,x,2) + y == x

f(x) = desolve(myequation, y, [10,2,1] )
print "f(x) = ", f(x)

fprime(x) = diff( f(x), x )
fprimeprime(x) = diff( fprime(x), x )

print "f(10) = ", N(f(10))
print "f'(10) = ", fprime(10)
print fprimeprime(x) + f(x) - x
```

That code above produces the output below:

```
f(x) =  x - 8*cos(10)*cos(x)/(cos(10)^2 + sin(10)^2) -
          8*sin(10)*sin(x)/(cos(10)^2 + sin(10)^2)
f(10) =  2.00000000000000
f'(10) =  1
0
```

As you can see, the output above verifies our three requirements: first, that $f(10) = 2$; second, that $f'(10) = 1$; third, that $f(x)$ satisfies the differential equation $f''(x) + f(x) = x$.

4.22.3. Graphing a Slope Field

Suppose I'd like to solve the differential equation:

$$\frac{dy}{dx} + \frac{y}{x} + x = 2$$

with initial conditions $y(2) = 4$. However, I'd like to know how the solutions would vary under other initial conditions also. First, I should solve the initial-value problem as we did in the previous subsection. I should type the following code:

```
y = function('y',x)
myequation = diff(y,x) + y/x + x == 2
h = desolve(myequation, y, [2, 4] )

print h.expand()
```

Now I have the answer:

```
-1/3*x^2 + x + 20/3/x
```

which really means

$$f(x) = \frac{-1}{3}x^2 + x + \frac{20/3}{x}$$

and therefore

$$f'(x) = \frac{-2}{3}x + 1 - \frac{20/3}{x^2}$$

as well as

$$\frac{f(x)}{x} = \frac{-1}{3}x + 1 + \frac{20/3}{x^2}$$

allowing us to conclude that the solution is correct by checking:

$$\begin{aligned} f'(x) + \frac{f(x)}{x} + x &= \left(\frac{-2}{3}x + 1 - \frac{20/3}{x^2}\right) + \left(\frac{-1}{3}x + 1 + \frac{20/3}{x^2}\right) + x \\ &= \left(\frac{-2}{3} + \frac{-1}{3} + 1\right)x + (1+1) + \left(\frac{-20/3}{x^2} + \frac{20/3}{x^2}\right) \\ &= 0 + 2 + 0 = 2 \end{aligned}$$

The new step we're going to take is that we're going to solve the original differential equation for dy/dx. In this case, we obtain

$$\frac{dy}{dx} = -\frac{y}{x} - x + 2$$

which means

$$\frac{dy}{dx} = \left(\frac{1}{3}x - 1 - \frac{20/3}{x^2}\right) - x + 2 = \frac{-2}{3}x - \frac{20/3}{x^2} + 1$$

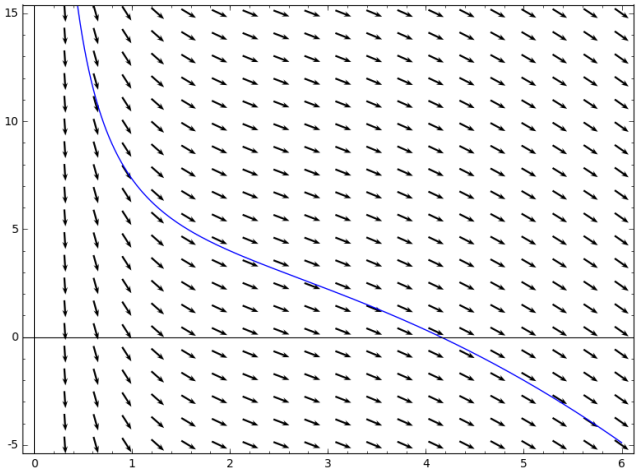
and note that *all solutions* to the differential equation must obey that relationship, not just the one that satisfies our initial conditions.

Next, we're going to construct a slope field plot. For every position on the coordinate plane, we can use the equation above to find what dy/dx should be. Sage will select about 400 points, in a 20×20 pattern, forming a

grid over the coordinate plane. For each point, Sage will compute dy/dx at that point, and will draw a vector with that slope, centered on that point. Our plot will cover $0 < x < 6$ and $-5 < y < 15$, and we will also plot our particular $f(x)$ from the above, superimposed on the same image. Here is the code:

```
var("y")
f(x) = -1/3*x^2 + x + (20/3)/x

P1 = plot_slope_field( (-2/3)*x - (20/3)/x^2 + 1, (x, 0, 6),
         (y, -5, 15), headlength = 4, headaxislength=3 )
P2 = plot( f(x), (x, 0, 6), ymax=15, ymin=-5 )
P = P1 + P2
P.show()
```

That code produces the following plot:



As you can see, our solution (the solid curve) matches all the vectors that are very near to it. In some ways, our solution is like connecting a sequence of adjacent vectors, head-to-tail, to get a sweeping curve that takes us from $x = 0$ to $x = 6$. Likewise, all other solutions would be similar sweeping curves. If you had a different initial condition, you'd start at a different point, but you'd still connect one vector to another, head-to-tail, to get a sweeping curve.

A brief word is in order about the two options

```
headlength = 4, headaxislength = 3
```

Experts in differential equations prefer to not have arrowheads on their vectors when making slope field plots. I guess this makes sense, because they might crowd the image (especially if the vectors are close together). Furthermore, the arrowhead always goes on the right of the line segment representing the shaft, so the arrowhead actually conveys zero additional information. On the other hand, I really do prefer the arrowheads to be there, because they are visually appealing and because they are useful to

students who are just beginning their exploration of differential equations. The reader might want to experiment by backspacing over those two options to see how the image changes when those options are deleted.

4.22.4. The Torpedo Problem: Working with Parameters

Let's suppose that I wanted to solve the differential equation for a torpedo accelerating underwater, from a deep submarine up to the surface. Of course, the force of hydrodynamic drag will not be negligible, and I have to include it. The weight of the torpedo and the thrust of the engine would be important characteristics as well. First we start with a general equation

$$ma = \Sigma F = \underbrace{-kv}_{\text{drag}} - \underbrace{mg}_{\text{weight}} + \underbrace{T}_{\text{thrust}}$$

where m is the mass of the torpedo, k is some coefficient of drag, $g = 9.82$ m/s^2 is the acceleration due to gravity, and T is the thrust of the torpedo's engine. Next, I can make this a differential equation by making altitude $y(t)$, the velocity $y'(t)$ and the acceleration $y''(t)$. (By the way, I use the convention that "up" is positive and "down" is negative.) At this point, I have

$$my''(t) = \underbrace{-ky'(t)}_{\text{drag}} - \underbrace{mg}_{\text{weight}} + \underbrace{T}_{\text{thrust}}$$

which we will now enter into Sage. We will analyze the code below, line by line.

```
var("m k T g t")
y = function('y', t)

general_equation=  m*diff(y, t, 2) == -k*diff(y, t) - m*g + T
my_equation = general_equation(m=1000, k=0.5, T=20000, g=9.82)

f(t) = desolve( my_equation, y, [0, -2000, 0] )
print "y(t) = ", f(t)

plot( f(t), 0, 20, gridlines="minor" )
```

The first line there declares five variables, as we've seen throughout this book. The second line declares that the function $y(t)$, the altitude at time t, is unknown for now. Then you have a faithful reproduction of the governing differential equation as the variable `general_equation`, in the third line.

The fourth line substitutes 1000 kg for the mass of the torpedo, 0.5 for the coefficient of drag, 20,000 N for the thrust, and the usual 9.82 m/s^2 for g. Confusingly, the units on the coefficient of drag are kg/s, because that's how we can get Newtons after multiplying that coefficient by a velocity in m/s.

Having the assignments of the parameters at the top of the program allows us to experiment with different values of those parameters, to see the impact on the final solution. Shortly we will add that the torpedo has initial depth of 2000 m and initial velocity of 0 m/s.

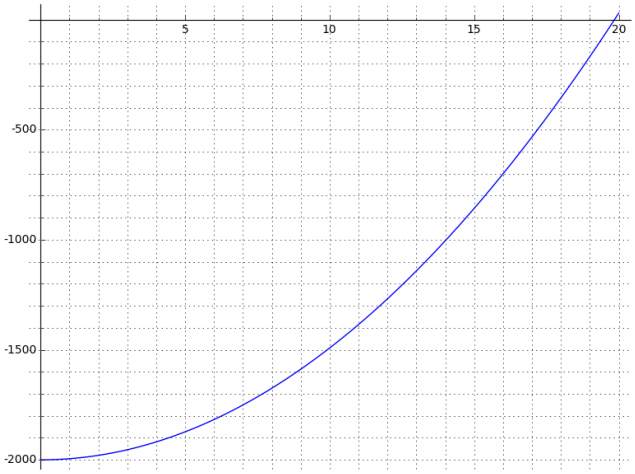
The fifth line, no doubt familiar by now, will use `desolve` to solve this differential equation. We give the initial condition that at $t = 0$, the altitude is $y = -2000$ m and the velocity is $dy/dt = 0$ m/s. The answer gets stuffed into $f(t)$.

By the time we reach the sixth line, $f(t)$ is known, because `desolve` computed it. The need for two functions, $y(t)$ and $f(t)$ confuses some students. Think of $y(t)$ as a placeholder for where the final answer will go, but $f(t)$ is the actual answer. For example, in $3x + 5 = 5x - 3$, the placeholder for the final solution is x but the actual answer is 4.

In any case, the solution we get is

```
y(t) =  20360*t + 40720000*e^(-1/2000*t) - 40722000
```

and in addition, we get a very nice plot:



If you look at the graph closely, you will see that for the rightmost 25% of the time, the graph looks like a line. Also, for large values of t, our function $y(t)$ will be a line. Without the force of hydrodynamic drag, however, our function $y(t)$ would be a parabola. Therefore, we can see that the force of drag does not merely change the solution quantitatively, but rather qualitatively. The slope of that line is called "the terminal velocity" and at that velocity, the force of drag is equal to the thrust. Of course, it should be noted that for $y > 0$ the torpedo is above the water and therefore a completely different model must be used at that stage.

4.23. Laplace Transforms

The Laplace Transform, named for Pierre-Simon de Laplace (1749–1827), is delightful in so many ways. First, it renders many difficult problems in

Differential Equations into mere *College Algebra* problems, though often rather complicated ones. Second, it is an amazing analogy between two worlds—the world of time, t, our world—and the world of transform space, s, which is an alien world to our thinking, yet one that we can use to solve important and practical problems. Third, to compute a Laplace transform by hand is a great review of the techniques of integration—offering a review of *Calculus II*, typically a year or more after it was taught. Let's now see how to compute Laplace Transforms in Sage.

4.23.1. Transforming from $f(t)$ to $L(s)$

For our opening example, we will find the $L(s)$ that matches up with $f(t) = t\cos(t)$ in our world. We would type

```
var("s")
f(t) = t*cos(t)
L(s) = laplace(f, t, s)
print L(s)
```

and we receive back the response:

```
2*s^2/(s^2 + 1)^2 - 1/(s^2 + 1)
```

Let's just change $f(t)$ to a new function, keeping the other four lines the same. Now we consider $f(t) = 5te^{t-3}$ as our example. We change only the second line to the following:

```
f(t) = 5*t*exp(t-3)
```

which results in the response:

```
5*e^(-3)/(s - 1)^2
```

After all these years, it is still shocking to me how an *enormous* collection of functions, some of which are rather complex in terms of t, merely become rational functions of s. Of course, that's the whole point: if the functions did not become vastly simpler in the transform world, then there would be no reason to transform them in the first place!

4.23.2. Computing the Laplace Transform "The Long Way"

Because the Laplace Transform is defined as

$$L(s) = \int_{0^+}^{\infty} f(t)e^{-st}\,dt$$

we normally would imagine that the code for computing the Laplace Transform of our previous example would be as below:

```
var("s")
f(t) = 5*t*exp(t-3)
L(s) = integral( f(t)*exp(-s*t), t, 0, oo )
print L(s)
```

However, we get an error from Sage, that consists of a lot of uninterpretable lines, followed by the following text as the last two lines. Remember: in Sage, the last lines of an error message are what are most important.

```
ValueError: Computation failed since Maxima requested additional
constraints; using the 'assume' command before integral evaluation
*may* help (example of legal syntax is 'assume(s-1>0)', see
`assume?` for more details)
Is s-1 positive, negative or zero?
```

As you can see, Maxima, the package inside of Sage that computes integrals, is confused about the sign of $s > 1$. Therefore, we insert the line `assume(s>1)` immediately above the line containing the integral. The first time you saw the `assume` command was probably on Page 60, and while there might be other uses for it, I've only ever seen it used to help the integration and summation algorithms reach a conclusion. It is very rare. Another example, in a summation, can be found on Page 184.

After doing this, we get the result

```
5*e^(-3)/(s - 1)^2
```

which is undoubtedly correct. However, out of curiosity, we might want to try the other two cases, namely `assume(s<1)` and `assume(s==1)`. However, we get the error message below in that case (after a lot of meaningless lines).

```
ValueError: Integral is divergent.
```

In summary, we feel as though Maxima is justified in demanding that explicit assumption, because the integral was not convergent without that assumption. Furthermore, the `laplace` command does this all for you, which is a nice convenience.

4.23.3. Transforming from $L(s)$ to $f(t)$

Let's now see how to convert our $L(s)$ from transform space back into our world. The code to be typed is

```
var("s t")
L(s) = 5*e^(-3)/(s - 1)^2
print inverse_laplace( L(s), s, t )
```

That code produces the correct answer:

```
5*t*e^(t - 3)
```

Alternatively, we can try challenging Sage with a harder problem. Imagine after a *Control Systems Engineering* problem that you have $L(s) = 1/(s^3 + 1)$ as your final answer. Computing the inverse transform is going to be a bit challenging by hand, because you have to do a partial fraction decomposition. Sage, however, can compute it easily. We simply change the middle line of the above code to

```
L(s) = 1/(s^3+1)
```

That produces the answer given below:

```
1/3*(sqrt(3)*sin(1/2*sqrt(3)*t) - cos(1/2*sqrt(3)*t))*e^(1/2*t)
        + 1/3*e^(-t)
```

However, that function is very hard to comprehend. For example, if I have to put this function into a technical mathematical paper, then I have to figure out how I'm going to encode it into LaTeX. Luckily, Sage can help with this. If I replace the last line with

```
latex( inverse_laplace( L(s), s, t ) )
```

Then I get the frightening response

```
\frac{1}{3} \, {\left(\sqrt{3} \sin\left(\frac{1}{2} \, \sqrt{3} t
\right) - \cos\left(\frac{1}{2} \, \sqrt{3} t\right)\right)}
e^{\left(\frac{1}{2} \, t\right)} + \frac{1}{3} e^{\left(-t\right)}
```

which I can insert into my mathematical paper in LaTeX. It encodes into the following:

$$f(t) = \frac{1}{3}\left(\sqrt{3}\sin\left(\frac{1}{2}\sqrt{3}t\right) - \cos\left(\frac{1}{2}\sqrt{3}t\right)\right)e^{\left(\frac{1}{2}t\right)} + \frac{1}{3}e^{(-t)}$$

which is much more readable.

4.24. Vector Calculus in Sage

In this section, we'll learn how to use Sage to compute the more important[9] operators in *Vector Calculus* including the divergence, the curl, the Laplacian, the Hessian and the Jacobian, as well as how to have Sage compute double integrals. It might be useful to review the material on Contour plots (Section 3.5 on Page 104), and the material on Vector Field plots (Section 3.7 on Page 114) as that will permit you to graph the many examples throughout this section.

Please do not feel bad if you find this material confusing. While physics faculty tend to know this material very well, most mathematics faculty to not have these formulas at their finger tips, unless they have taught the course recently or it is their research area. The phenomenally famous study guide for teaching this material to engineers is a book called *Div, Grad, Curl, and All That: an Informal Text on Vector Calculus*, by Harry Moritz Schey, published by W. W. Norton and Company. The fourth edition was published in 2004, but the first edition was published in 1973. The longevity of the popularity of this textbook is a testament to its uniquely chatty, almost sarcastic, conversational tone.

[9]Of course, the gradient is the most important operator in *Vector Calculus*, (at least in applied mathematics). Usually the gradient is introduced in an earlier course, such as *Multivariate Calculus*, but we learned how to compute gradients in Section 4.3.2 on Page 135. Therefore, we do not repeat it here.

4.24.1. Notation for Vector-Valued Functions

Before we start, we are going to use the following notation throughout this section. A mathematical function f that takes an n-dimensional vector as input, and which outputs an m-dimensional vector, is written $\vec{f} : \mathbb{R}^n \to \mathbb{R}^m$. Notice that this notation is consistent with linear algebra. If $\vec{f}(\vec{x}) = A\vec{x}$ for some $m \times n$ matrix A, then $\vec{f} : \mathbb{R}^n \to \mathbb{R}^m$.

The functions that we see in most of *Multivariate Calculus* would be those like

$$g(x, y, z) = x^2 + y^3 + z^4 + xyz^2$$

which have a multivariable (or vector) input, but a single output. This is indicated by $g : \mathbb{R}^3 \to \mathbb{R}$. In contrast, the functions we learn about during *Calculus I* are functions like $b(x) = 2e^{-x}$ and those are written as $b : \mathbb{R} \to \mathbb{R}$.

The least common case, which is not all that rare, is a function with one input and many outputs (or a vector output). An example of that would be a function that gives the position of an aircraft in $\mathbb{R}^3$ at any point in time, such as

$$\vec{s}(t) =< 200 + 3t,\ 300 - t,\ 400 + 5t >$$

which would be denoted $\vec{s} : \mathbb{R} \to \mathbb{R}^3$.

In summary, the dimensions of the input go on the first $\mathbb{R}$ and the dimensions of the output go on the second $\mathbb{R}$. However, the dimension "1" is never written. The hat on $\vec{f}$ and $\vec{s}$ were included because $\vec{f}$ and $\vec{s}$ have multidimensional (or vector) outputs and it was excluded on g and b because g and b have one-dimensional (or scalar) outputs.

4.24.2. Computing the Hessian Matrix

The Hessian[10] of a function $\mathbb{R}^3 \to \mathbb{R}$ is given by

$$H = \begin{bmatrix} \frac{\partial^2 f}{\partial x \partial x} & \frac{\partial^2 f}{\partial x \partial y} & \frac{\partial^2 f}{\partial x \partial z} \\ \frac{\partial^2 f}{\partial y \partial x} & \frac{\partial^2 f}{\partial y \partial y} & \frac{\partial^2 f}{\partial y \partial z} \\ \frac{\partial^2 f}{\partial z \partial x} & \frac{\partial^2 f}{\partial z \partial y} & \frac{\partial^2 f}{\partial z \partial z} \end{bmatrix}$$

and consists of all possible second partial derivatives, arranged in a 3×3 matrix. For other dimensions, the definition is modified in the obvious way. We will continue to use our example from the gradient section on Page 135. Specifically, we consider

$$g(x, y) = xy + \sin(x^2) + e^{-x}$$

For this example, the following code:

```
g(x,y) = x*y + sin(x^2) + e^(-x)
diff( g, 2 )
```

[10] Named for Ludwig Otto Hesse (1811–1874), who is known for results in mathematics and physics, and pioneering the study of applications of the determinant, but also for writing a popular elementary textbook on the analytical geometry of lines and circles.

produces the response:

```
[(x, y) |--> -4*x^2*sin(x^2) + 2*cos(x^2) + e^(-x)           (x, y) |--> 1]
[                                  (x, y) |--> 1           (x, y) |--> 0]
```

This odd format is to remind you that for each point in the coordinate plane (x, y), each of the second partial derivatives is a number. However, there are four of those, so one obtains four separate numbers. The natural arrangement is as a 2×2 matrix.

Another way of asking Sage to compute the Hessian Matrix is with the following code:

```
g(x,y) = x*y + sin(x^2) + e^(-x)
H = diff( g, 2 )

print H(0,3)
print
print H(0,0)
print
print H(sqrt(pi), 2)
```

As you can see here, we've asked for the Hessian but in a numerical sense. We've asked for the specific matrix at the points $(0, 3)$, $(0, 0)$, and $(\sqrt{\pi}, 2)$. The responses are what we expect, as noted below:

$$\begin{bmatrix} 3 & 1 \\ 1 & 0 \end{bmatrix} \quad \begin{bmatrix} 3 & 1 \\ 1 & 0 \end{bmatrix} \quad \begin{bmatrix} e^{-\sqrt{\pi}} - 2 & 1 \\ 1 & 0 \end{bmatrix}$$

Observe that three out of the four entries in the matrix (all but the upper-left corner) are locked at fixed values. The upper-left corner varies with x but not with y. Try other inputs to verify that this is true. Now if you look at the symbolic Hessian, you'll see that indeed, three of the four entries are constants, and the upper-left corner is a function of x but constant with respect to y.

Sometimes we just want to know the determinant of the Hessian. In mathematical economics, there are times where producing example functions $f(x, y)$ with one unique local (and therefore global) minimum is very useful.

When we restrict consideration to functions of two variables, $f(x, y)$ there is a very useful shortcut. It turns out that this phenomena (namely, that the local minimum or local maximum is unique) is guaranteed to occur if the determinant of the Hessian is positive for all points (x, y). This is the analogous result that univariate functions $g(x)$ have a unique local (and therefore global) minimum if $g''(x) > 0$ for all x. Furthermore, univariate functions $g(x)$ have a unique local (and therefore global) maximum if $g''(x) < 0$ for all x. With that in mind, we can type

```
g(x,y) = (x + y + 2)*(x + y + 1)*(x + y - 1)*(x + y - 2)
h(x,y) = det(diff( g, 2 ))
print h(x,y)
```

to get that determinant of the Hessian very quickly. In this case, the determinant is zero everywhere.

4.24.3. Computing the Laplacian

The Laplacian of a function $g : \mathbb{R}^3 \to \mathbb{R}$ is denoted either $\nabla \cdot \nabla g$ or $\nabla^2 g$, depending on which textbook you are reading, and is defined as

$$\nabla \cdot \nabla g = \nabla^2 g = \frac{\partial^2}{\partial x \partial x} g + \frac{\partial^2}{\partial y \partial y} g + \frac{\partial^2}{\partial z \partial z} g$$

which is the sum of the entries of the main diagonal of the Hessian matrix of g. The Laplacian is named for Pierre-Simon de Laplace (1749–1827).

Essentially, this is the sum of all second partial derivatives, *but excluding* the mixed partial derivatives. This operator is not built into Sage, but it is easy to compute anyway, using existing commands. Consider the function $g : \mathbb{R}^3 \to \mathbb{R}$ that we had seen on Page 204, namely

$$g(x, y, z) = x^2 + y^3 + z^4 + xyz^2$$

The relevant second partial derivatives are

$$\begin{aligned} \frac{\partial^2 g}{\partial x \partial x} &= 2 \\ \frac{\partial^2 g}{\partial y \partial y} &= 6y \\ \frac{\partial^2 g}{\partial z \partial z} &= 12z^2 + 2xy \end{aligned}$$

Therefore, the Laplacian of g is just

$$\nabla \cdot \nabla g = \nabla^2 g = 2 + 6y + 12z^2 + 2xy$$

and was easily computed by hand. However, for more complicated functions, we might prefer to use Sage. For this function g, we would type

```
g(x, y, z) = x^2 + y^3 + z^4 + x*y*z^2

L(x, y, z) = diff(g, x, x) + diff(g, y, y) + diff(g, z, z)

print "Laplacian:"
print L(x,y,z)

print "Hessian:"
print diff( g, 2 )
```

We obtain the following output:

```
Laplacian:
2*x*y + 12*z^2 + 6*y + 2
Hessian:
[     (x, y, z) |--> 2    (x, y, z) |--> z^2          (x, y, z) |--> 2*y*z]
[   (x, y, z) |--> z^2    (x, y, z) |--> 6*y          (x, y, z) |--> 2*x*z]
[ (x, y, z) |--> 2*y*z  (x, y, z) |--> 2*x*z (x, y, z) |--> 2*x*y + 12*z^2]
```

As you can see, the Laplacian is just the sum of the entries on the main diagonal of the Hessian.

4.24.4. The Jacobian Matrix

Now we're going to consider a function $\vec{f} : \mathbb{R}^3 \to \mathbb{R}^3$. One way to think of this vector-valued function

$$\vec{f}(x, y, z) = y\vec{i} + x^2\vec{j} + 3\vec{k} =< y,\ x^2,\ 3 >$$

is to think of it as three separate functions, each of which is $f_i : \mathbb{R}^3 \to \mathbb{R}$. In this example, those functions would be

$$\begin{aligned} f_1(x, y, z) &= y \\ f_2(x, y, z) &= x^2 \\ f_3(x, y, z) &= 3 \\ \vec{f}(x, y, z) &= < f_1(x, y, z),\ f_2(x, y, z),\ f_3(x, y, z) > \end{aligned}$$

As you can see, the job of each of f_1, f_2, and f_3 is to tell us one particular coordinate of $\vec{f}$. In this case, it is easy to compute the Jacobian[11] Matrix, which is defined as follows

$$J = \begin{bmatrix} \frac{\partial f_1}{\partial x} & \frac{\partial f_1}{\partial y} & \frac{\partial f_1}{\partial z} \\ \frac{\partial f_2}{\partial x} & \frac{\partial f_2}{\partial y} & \frac{\partial f_2}{\partial z} \\ \frac{\partial f_3}{\partial x} & \frac{\partial f_3}{\partial y} & \frac{\partial f_3}{\partial z} \end{bmatrix}$$

In this example, we would have

$$J = \begin{bmatrix} 0 & 1 & 0 \\ 2x & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

for the Jacobian Matrix. Sometimes when people use the word "Jacobian," they just mean the determinant of the Jacobian matrix. In this case, the answer would just be 0, because the matrix J has a row of all zeros, and is therefore obviously singular.

To calculate all these things in Sage, we would use the following code:

```
f(x,y,z) = [ y, x^2, 3 ]

print "Function:"
print f
print "Jacobian:"
print diff(f)
print "Jacobian Determinant:"
print det( f.diff() )
```

[11] Named for Carl Gustav Jacob Jacobi (1804–1851).

As you can see, Sage considers the Jacobian to be the natural meaning of the word "derivative" for a function that has a vector input and a vector output. Upon further thought, this makes sense. Specifically, if the output of some function is one-dimensional, but with a multidimensional input, then the Jacobian would be equal/identical to the gradient as a row vector, and Sage considers the gradient to be the natural meaning of the word "derivative" for functions for many-input one-output functions.

However, if the determinant of the Jacobian Matrix is desired, one must explicitly use the `det` command, like any other determinant.

4.24.5. The Divergence

The divergence is an important operator, but it is not built into Sage. We saw that the Laplacian operator was not built into Sage either, and we will see shortly that the curl is also not built in to Sage. However, the divergence is extremely easy to calculate using existing commands.

For any function $\vec{f}: \mathbb{R}^3 \to \mathbb{R}^3$, the divergence is defined by

$$\nabla \cdot \vec{f} = \text{div}\vec{f} = \frac{\partial f_1}{\partial x} + \frac{\partial f_2}{\partial y} + \frac{\partial f_3}{\partial z}$$

which can be thought of as the sum of the entries on the main diagonal of the Jacobian matrix. (Sometimes this is called "the trace" of the Jacobian matrix.) In that sense, it is very similar to the Laplacian, which is the sum of the entries on the main diagonal of the Hessian matrix.

Observe that for every point in ordinary 3-dimensional space, the divergence will provide us with a number, not a vector. This means that the divergence of $\vec{f}$ is a function $\mathbb{R}^3 \to \mathbb{R}$.

Let's now compute an example. Consider the rather fanciful function

$$\vec{f}(x, y, z) = xy^2\vec{i} + yz^2\vec{j} + zx^2\vec{k}$$

where we will compute both the divergence and the Jacobian matrix. The code is below. Since this is an easy example, you might want to compute it with pencil and paper first.

```
f1(x,y,z) = x*y^2
f2(x,y,z) = y*z^2
f3(x,y,z) = z*x^2

f(x, y, z) = ( f1(x,y,z),  f2(x,y,z),  f3(x, y, z) )

div(x, y, z) = diff( f1, x ) + diff( f2, y ) + diff( f3, z )

print "Divergence:"
print div(x,y,z)

print "Jacobian:"
print derivative( f )
```

That code produces this output:

```
Divergence:
x^2 + y^2 + z^2
Jacobian:
[   (x, y, z) |--> y^2 (x, y, z) |--> 2*x*y     (x, y, z) |--> 0 ]
[     (x, y, z) |--> 0   (x, y, z) |--> z^2 (x, y, z) |--> 2*y*z ]
[ (x, y, z) |--> 2*x*z     (x, y, z) |--> 0   (x, y, z) |--> x^2 ]
```

As you can see, the divergence is equal to sum of the entries of the main diagonal of the Jacobian matrix.

4.24.6. Verifying an Old Identity

You might have been taught the theorem that the divergence of the gradient is equal to the Laplacian. On Page 206, we computed the Laplacian of

$$g(x, y, z) = x^2 + y^3 + z^4 + xyz^2$$

Therefore, let's now compute the divergence of its gradient, and see if we get the same answer that we got earlier. To compute the gradient, we type

```
g(x, y, z) = x^2 + y^3 + z^4 + x*y*z^2

print "Original:"
print g(x, y, z)

gradient = derivative( g )

print "Gradient:"
print gradient
```

which results in the output:

```
Original:
x*y*z^2 + z^4 + y^3 + x^2
Gradient:
(x, y, z) |--> (y*z^2 + 2*x, x*z^2 + 3*y^2, 2*x*y*z + 4*z^3)
```

Now with that in mind, we can interpret the three components of the gradient as f_1, f_2, and f_3. We will program Sage to take the partial derivatives and compute the sum. Use the following code:

```
f1(x,y,z) = y*z^2 + 2*x
f2(x,y,z) = x*z^2 + 3*y^2
f3(x,y,z) = 2*x*y*z + 4*z^3

Lap(x,y,z) = diff( f1, x ) + diff( f2, y ) + diff( f3, z )
print "Laplacian:"
print Lap(x, y, z)
```

Finally, we get the output:

```
Laplacian:
2*x*y + 12*z^2 + 6*y + 2
```

As you can see, this is an exact match for the Laplacian that we had computed on Page 206. We have now verified that the divergence of the gradient (of this particular function) is equal to the Laplacian (of this particular function). That's a far cry from proving the theorem in general, but it is nice to know that we can at least verify individual instances.

4.24.7. The Curl of a Vector-Valued Function

The curl of a function $\vec{f}: \mathbb{R}^3 \to \mathbb{R}^3$, of the form

$$\vec{f}(x,y,z) =< f_1(x,y,z),\ f_2(x,y,z),\ f_3(x,y,z) >$$

is denoted $\nabla \times \vec{f} = \text{curl}\vec{f}$ and is computed by

$$\nabla \times \vec{f} = \text{curl}\vec{f} = \left(\frac{\partial}{\partial y}f_3 - \frac{\partial}{\partial z}f_2\right)\vec{i} + \left(\frac{\partial}{\partial x}f_1 - \frac{\partial}{\partial z}f_3\right)\vec{j} + \left(\frac{\partial}{\partial x}f_2 - \frac{\partial}{\partial y}f_1\right)\vec{k}$$

which is a formula that surely is not very easy to memorize. Accordingly, there is a trick to remembering it.

A Trick for Remembering the Curl Formula:

Personally, I've always been frustrated that the curl is so hard to calculate, particularly because it is the most important operator after the gradient in any engineering or physics applications. There is actually a semi-nonsense memory hook for remembering how to take the curl when doing mathematics by hand. Consider

$$\begin{aligned}
\det\begin{bmatrix} \vec{i} & \vec{j} & \vec{k} \\ \frac{\partial}{\partial x} & \frac{\partial}{\partial y} & \frac{\partial}{\partial z} \\ f_1 & f_2 & f_3 \end{bmatrix} &= \\
&= \det\begin{bmatrix} \frac{\partial}{\partial y} & \frac{\partial}{\partial z} \\ f_2 & f_3 \end{bmatrix}\vec{i} - \det\begin{bmatrix} \frac{\partial}{\partial x} & \frac{\partial}{\partial z} \\ f_1 & f_3 \end{bmatrix}\vec{j} + \det\begin{bmatrix} \frac{\partial}{\partial x} & \frac{\partial}{\partial y} \\ f_1 & f_2 \end{bmatrix}\vec{k} \\
&= \left(\frac{\partial}{\partial y}f_3 - \frac{\partial}{\partial z}f_2\right)\vec{i} - \left(\frac{\partial}{\partial x}f_3 - \frac{\partial}{\partial z}f_1\right)\vec{j} + \left(\frac{\partial}{\partial x}f_2 - \frac{\partial}{\partial y}f_1\right)\vec{k} \\
&= \text{curl}\vec{f}
\end{aligned}$$

Essentially, if you can remember that 3×3 matrix, and take its determinant successfully, then you can compute the curl of any function $\mathbb{R}^3 \to \mathbb{R}^3$.

You might be curious to know why I called it semi-nonsense. The reason is that $\partial/\partial x$ is an operator, and computing

$$\frac{\partial}{\partial x}f_2$$

is not a multiplication in any way, shape or form. Yet, the process of computing those 2×2 determinants is exactly and precisely two multiplications

separated by a subtraction. However, when following the above process blindly, the symbols do work out to be in the correct spots visually, and therefore the memory hook is useful—even though it doesn't really make sense mathematically. This memory hook is the university-level equivalent to the middle-school joke that (3)(3) should really equal 33.

A Simple Example of the Curl:

Sometimes a simple example is best. The following example was found in the Wikipedia article "Curl: (mathematics)" on June 29th, 2014. While I normally would never cite Wikipedia, the exposition is remarkably lucid in this case; it would be a pity to substitute it with a more complicated example, merely to avoid the stigma of Wikipedia. I have taken extra care to make sure that there are no errors in the example, as Wikipedia is famously error-prone.

Consider the following function:

$$\vec{c} = y^2\vec{i} =< y^2, 0, 0 >$$

Since f_2 and f_3 are the zero function in this case, then we can focus on f_1. We can see that

$$\frac{\partial}{\partial x}f_1 = 0 \qquad \frac{\partial}{\partial y}f_1 = 2y \qquad \frac{\partial}{\partial z}f_1 = 0$$

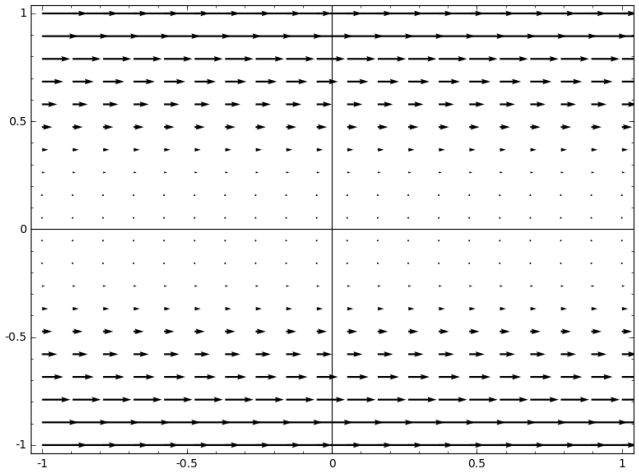
and thus the normal formula for the curl simplifies rapidly

$$\begin{aligned} \nabla \times \vec{f} = \text{curl}\vec{f} &= \left(\frac{\partial}{\partial y}f_3 - \frac{\partial}{\partial z}f_2\right)\vec{i} + \left(\frac{\partial}{\partial x}f_1 - \frac{\partial}{\partial z}f_3\right)\vec{j} + \left(\frac{\partial}{\partial x}f_2 - \frac{\partial}{\partial y}f_1\right)\vec{k} \\ &= (0-0)\vec{i} + (0-0)\vec{j} + (0-2y)\vec{k} \\ &= -2y\vec{k} =< 0, 0, -2y > \end{aligned}$$

Let's look at a vector field plot. In order to plot it on a two-dimensional page, we have to ignore the z-coordinate entirely. That's fine in this case, because our example does not use z. We can type

```
var("y")
plot_vector_field( (y^2,0), (x,-1,1), (y,-1,1) )
```

We obtain the image:



As you can see, all the arrows point to the right. The x-coordinate does not matter; the arrows near the x-axis (those with a small y-coordinate, in absolute value) have a small magnitude, while those far from the x-axis (those with a large y-coordinate, in absolute value) have a large magnitude. Now ask yourself, if this plot was representing a flowing fluid, how would you expect a paddle-wheel to behave if inserted:

- above the x-axis?
- on the x-axis?
- below the x-axis?

Take a moment to really think about this. (Please do not read further until you've tried to answer that thought experiment.)

In any case, if the paddle wheel is on the x-axis, there would be no rotation. However, above the x-axis, we would expect clockwise rotation, and below the x-axis, we would expect counter-clockwise rotation. That's because one side of the paddle wheel is being buffeted to much greater extent than the other. Recalling that clockwise rotation on a two-dimensional piece of paper indicates a vector into the paper, and counter-clockwise rotation indicates a vector out of the paper, we see that our answer of $< 0, 0, -2y >$ makes sense.

- When y is zero, all coordinates of the curl are zero, thus the curl is zero. (That means no paddle wheel rotation.)
- When y is negative, the third coordinate is positive, thus the curl points out of the paper. (That means counter-clockwise paddle wheel rotation.)
- When y is positive, the third coordinate is negative, thus the curl points into the paper. (Similarly, that means clockwise paddle wheel rotation.)

Now we must compute this in Sage. The calculus was easy in this extremely simple example, but in other examples it might be extremely unpleasant. The following code-snippet is very handy:

```
var("y z")

f1 = y^2
f2 = 0
f3 = 0

curl = ( diff(f3, y) - diff(f2, z), diff(f1, z) - diff(f3, x),
          diff(f2, x) - diff(f1, y)  )
print curl
```

Naturally, we get the expected output of `(0, 0, -2*y)` from that code.

A Harder Example of the Curl:

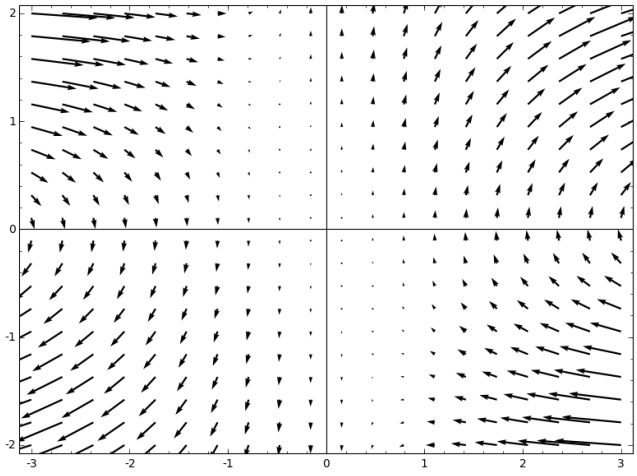
Now let's consider a harder example, where we would not want to compute the curl by hand. A slightly more complicated function is

$$\vec{f}(x, y, z) =< x^2 y,\ x + y + \sin(x),\ 0 >$$

for which we can use the following code:

```
var("y z")

f1 = x^2*y
f2 = x + y + sin(x)
f3 = 0

curl = ( diff(f3, y) - diff(f2, z), diff(f1, z) - diff(f3, x),
          diff(f2, x) - diff(f1, y)  )
print curl

fvec = (f1, f2)
plot_vector_field( fvec, (x, -3, 3), (y, -2, 2) )
```

As you can see, we added two lines to make a vector plot, and obtained the following image:



4.24.8. A Challenge: Verifying Some Curl Identities

Here's a challenge for you. Open up a good textbook for *Vector Calculus*.

- Find several functions of the form $g : \mathbb{R}^3 \to \mathbb{R}$ and verify that the curl of the gradient is the zero vector, $\vec{0} =< 0, 0, 0 >$.
- Find several functions of the form $\vec{f} : \mathbb{R}^3 \to \mathbb{R}^3$ and verify that the divergence of the curl is indeed 0.

Note: The second one is a bit harder, since neither the curl nor the divergence is predefined in Sage. However, the code-snippet that I used earlier to compute two examples of the curl should be very useful.

4.24.9. Multiple Integrals

The process of integration in Sage is the same for multiple integrals as it is for individual integrals. However, there is an alternative notation that makes things easier to understand.

$$\int_0^1 xe^{-x^2}\,dx$$

can be written in Sage as

```
integral( x*exp(-x^2), x, 0, 1)
```

as was taught in Section 1.12 starting on Page 51. However, the following syntax is also acceptable

```
integral( x*exp(-x^2), (x, 0, 1) )
```

That second notation seems undesirable, because it has more characters and it has nested parentheses. However, we'll now see that it makes more sense when working with multiple integrals. Consider the following integral:

$$\int_0^1 \int_0^y e^{y^2}\,dx\,dy$$

In the old notation, this would be

```
var("y")
integral( integral( exp(y^2), x, 0, y), y, 0, 1)
```

However, in the new notation, we write

```
var("y")
integral( integral( exp(y^2), (x, 0, y) ), (y, 0, 1) )
```

To me, it seems easier to understand that the "inner" integral has $0 < x < y$ and the "outer" integral has $0 < y < 1$. This new notation is similar to the way of expressing intervals that we first saw on Page 104, under the heading "Backwards Compatibility," when learning about advanced plotting techniques.

Chapter 5

Programming in Sage and Python

As you can see from earlier chapters, many important mathematical tasks can be done in Sage without having too many lines of code. Often just a small number of commands will take care of whatever you need to have done. On the other hand, there are times when we seek to do something deeper and more complex. Elaborate tasks often require more code, ranging from a dozen lines or so, all the way up to enormous programs of 10,000 lines written by several developers scattered around the globe over a period of many years.

In this chapter, we'll learn how to write intermediate-sized programs in Sage, to do some interesting mathematical tasks. Since Sage is built on top of Python, we will be learning respectable percentage of Python along the way. Of course, Python is a major computer language and has many features, and we cannot hope to cover all of them here. However, we will see each of the most common Python commands used in a mathematically relevant way.

By the way, it is useful to note that this chapter is written assuming that you know the contents of Chapter 1, and how to use SageMathCloud, but not necessarily any of the content of the other chapters of this book. The suggested pace through this chapter is to study sections 5.1 through 5.7, each on their own separate evening. Perhaps patient and gifted students can do the sections two at a time, but even this is not recommended. This material is somewhat heavy, and is better digested in small chunks.

As the Sage examples grow longer, it might become tedious to type them, letter-by-letter, into your computer. Remember, you can instead cut-and-paste these lines into Sage—you do not have to retype the examples letter by letter! Simply highlight the code from the (free online) pdf-version of this book, give the copy command in your pdf viewer, go to a Sage window, and give the paste command.

Similarly, it is really important that you use SageMathCloud for this chapter, and not the Sage Cell Server. I use the Sage Cell Server almost all the time, and while the 10-line to 20-line code blocks that we'll write in this chapter can and do run well there, you'll want to keep a permanent record of your code for many reasons. First, if there is a small glitch in the network, it is a pity to lose all the code that you've been working on for an hour. Second, frequently when experts program, they will go back to some old code that they have written and cut-and-paste it into the new program. Third, if you forget how to do something, it is great to be able to look back at old code, and see how it was done. Because I am stubborn and overuse the Sage Cell Server (underusing SageMathCloud), I frequently lose my work and have to start over from scratch.

Last but not least, I have presented you with challenges throughout this chapter. Every once in a while, after two or three concepts have been introduced, I pose a problem that I ask you to solve yourself. Be sure to actually try these, as they will greatly enhance your ability to learn this important topic. The only way to learn how to program is to program. I am confident that you will find the examples relatively easy but useful and non-trivial. On the other hand, if you are not willing to try these challenges yourself, then there will be no benefit to you from reading this chapter—you might as well stop reading now.

5.1. Repetition without Boredom: The For Loop

Many of us have been taught that computers are best at automating repetitive identical (or very similar) tasks. This is great news, because it can save a human being from doing a tedious and boring task which might take ages to complete. The best example of this, and the most commonly used, is called the "for loop." We'll learn about that now.

5.1.1. Using Sage to Generate Tables

Let's say I wanted to generate a list of the first 100 perfect squares. The code for that would be

```
for i in range(0,100): j^2
```

and this means that i will taken on the values $0, 1, 2, 3, \ldots, 99$ but not 100. This way the `range(0,x)` will always have x items in it. That was an easy, but not terribly useful example. If I'm graphing by hand, I might want to know the value of a function at points inside $-10 \leq x \leq 10$. Let's say the function was $f(x) = x^3 - x$. Then I'd type

```
for x in range(-10,11):
   x^3-x
```

That was interesting, but not crucial, because we've learned how to make Sage graph directly, on Page 8. However, that is how computers

graph functions—they will evaluate a function at 1000 appropriate points that are very close together, and draw 1000 dots, one at each location. To the human eye, it looks like a very smooth and clear graph. It appears to be a nice sweeping curve, whereas in reality, it is just a collection of dots.

By the way, did you notice how the first loop, listing the perfect squares, was all on one line? Meanwhile, the loop about $x^3 - x$ was on two lines. While it is legal to leave everything all on one line, it is considered good style to use multiple lines. As we learn to make more and more complex loops, you'll see why. It is important that every bit of code in a computer program be readable. Otherwise, it is hard to repair or enhance the program. It is not sufficient that a computer program merely "just works."

Now let's see an example from finance. Let's say that I want to generate all possible final amounts for a single-deposit $ 15,000 investment, with the interest rate unknown, but the duration being 5 years, compounded annually. Suppose I want to consider all interest rates of the form 1%, 2%, 3%, 4%, etc..., up to 29%. The formula for compound interest is $A = P(1 + i)^n$, and therefore, I would code

```
for i in range(1,30):
    N( 15000*(1+(i/100))^5 )
```

The indentation, by the way, is important. Use the tab key for that. The indentation shows which commands are to be repeated by the `for` loop.

Note that if you leave out the `N()` then you get exact fractions (rational numbers) which looks really funny. Try removing the `N( )` and see what happens—but make sure each "`(`" has as corresponding "`)`" after the modification.

5.1.2. Carefully Formatting the Output

In any case, that code was kind of hard to read. Alternatively, I can type

```
for i in range(1,30):
   print i,
   print " implies ",
   print N( 15000*(1+(i/100))^5 )
```

A little bit of background information might help here. The indentation serves to show that the indented commands are subordinate to the `for` command. That means they will be repeated. A print command will display on the screen what ever comes after it, but the comma tells it not to break to a new line.

Now I recommend you try the following three variations on the above:

- try adding a comma at the end of the third `print` statement,
- try it without commas at all,
- try it with only the comma after "implies" removed, but the comma after `i` being present.

Let's make our code more powerful now. If I wanted to go every 1/4th of one percent, I could do

```
for i in range(1,120):
    print N(i/400),
    print " implies ",
    print N( 15000*(1+(i/400))^5 )
```

which entailed three changes. First, the `(i/100)` in the big `N()` became `i/400`, to make the steps 1/400 (a quarter of a percent) instead of 1/100 (or one percent). Next, the loop goes to 120 instead of 30, so that I stop at 29.75%, instead of stopping one-fourth of the way there. Lastly, I decided to make the leftmost print show the actual rate, rather than the value of i, so that went from `print i` to `print N(i/100)`. Leaving off the `N()` in that first print produces the funny result of Sage trying to reduce the fractions that comprise the interest rate into lowest terms. Try it yourself and see what I mean.

We can make our earlier example with $x^3 - x$ produce more professional-looking output too. We could type

```
for x in range(-10,11):
    print "x=",
    print x,
    print " means y=",
    print x^3-x
```

At this point, our code is getting rather "tall" in the sense that we're using up a large number of lines but not doing very much of anything. It turns out that we can save some vertical space by combining the print statements. This is permitted so long as we keep the commas. For example, give the following code a try.

```
for x in range(-10,11):
    print "x=", x, " means y=", x^3-x
```

5.1.3. Arbitrary Lists

Sometimes the list of values you want is kind of arbitrary. For example, if I wanted to evaluate $x^3 - x$ at the x-values 2, 5, 7/4, 3/10, and $-28/11$, then I would type

```
for x in [-2, 5, 7/4, 3/10, -28/11]:
    print "x=",
    print x,
    print " means y=",
    print x^3-x
```

As you can see, I just replaced `range(-10,11)` with the list `[-2, 5, 7/4, 3/10, -28/11]`. This is an example of how Sage uses the Python list construct. We've also seen that in numerous places throughout the book.

5.1.4. Loops with Accumulators

You can also give the loop some "memory" by storing information in a variable. The most common uses of that are to find the sum of the entries in some list, but you can also use it to find a minimum value, a maximum value, or the product of the entries. A very old term for adding things up by using a `for` loop, from the earliest days of computation, is "an accumulator." I find it a useful term, but to a computer scientist it will have an antique ring to it.

Consider the following snippet of code, which will compute the sum of the integers between 0 and 1000 (inclusively).

```
total = 0
for i in range(0, 1001):
    total=total+i
print total
```

Wasteful Loops:

It would be wasteful and unwise to instead print each number from 1 to 1000, and then add them up. The code to do that would be

```
total = 0
for i in range(0, 1001):
    print i
    total=total+i
print total
```

Note the "`print i`" between the "`for`" statement and the "`total = total+i`" statement. Here there is too much output to display and the list of numbers is enormous, scrolling downward for a long time. It is important to avoid such techniques (especially if using a million or a billion numbers) as important computational resources, such as bandwidth and computing time, would be thereby wasted. The waste is particularly exaggerated in a system like Sage because you are using the internet to transfer all those intermediate values from the Sage server to your web browser. When my students are frustrated that their code is running extremely slowly, often the cause is that they have superfluous `print` statements which are wasting time.

By the way, did you notice how the 0 was included, but the 1001 was excluded? This is similar to the examples in Section 5.1.1 on Page 218, where we evaluated $f(x) = x^3 - x$ over the domain `range(-10, 11)`, where -10 was included but 11 was excluded. The reason that Python works that way is so that `range(0, 10)` will run exactly 10 times, `range(0, 100)` will run exactly 100 times, and `range(0, 1000)` will run exactly 1000 times. By the way, each "pass" or "run" of a `for` loop is called "an iteration."

Computing this Sum Directly:

One last point should be made, from the point of view of pure mathematics. There is a formula for adding up all the numbers in an arithmetic progression. An arithmetic progression is a list of numbers where the difference between two consecutive numbers, anywhere in the list, is always some fixed constant. That fixed constant is called the common difference. For example, consider

$$3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, \ldots$$

or alternatively

$$72, 69, 66, 63, 60, 57, 54, 51, 48, 45, \ldots$$

as well as

$$1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, \ldots$$

As you can see, these have a common difference of +2, -3, and +1 respectively. The sum of an arithmetic progression is given by

$$S = \frac{(a+z)(n)}{2}$$

where a is the first member in the progression, z is the last member in the progression, n is the number of members, and S is the final sum. We can more directly compute our objective with

$$S = \frac{(1+1000)(1000)}{2} = (1001)(500) = 500,500$$

obtaining the same answer but with much less computational effort.

5.1.5. Using Sage to find a Limit, Numerically

Let's say that you wanted to investigate what the function

$$f(x) = e^{1/(8-x)}$$

is doing around the value $x = 8$. Then you could write the following code

```
for i in range(0,9):
   xtemp = 8 - 10^(-i)
   print N(  e^(  1/(  8-xtemp  )  )  )
```

which inquires about the values of $f(x)$ at the points

$$7, 7.9, 7.99, 7.999, 7.9999, 7.99999, 7.999999, 7.9999999, 7.99999999$$

Those numbers also have the names

$8-10^{0}$, $8-10^{-1}$, $8-10^{-2}$, $8-10^{-3}$, $8-10^{-4}$, $8-10^{-5}$, $8-10^{-6}$, $8-10^{-7}$, $8-10^{-8}$

Evaluating $f(x)$ there yields the following output

```
2.71828182845905
22026.4657948067
2.68811714181614e43
1.97007111401705e434
8.80681822566292e4342
2.80666336042612e43429
3.03321539680209e434294
6.59223253461844e4342944
1.54997674664843e43429448
```

which is clearly going to infinity. (If $1.54\cdots\times10^{43,429,448}$ isn't "close enough" to infinity, then I don't know what is!) Then by changing the $8-10^{-i}$ into $8+10^{-i}$ we would obtain

```
for i in range(0,9):
    xtemp=8+10^(-i)
    print N(  e^(  1/(  8-xtemp  )  )  )
```

which inquires about the values of $f(x)$ at the points

$$9, 8.1, 8.01, 8.001, 8.0001, 8.00001, 8.000001, 8.0000001, 8.00000001$$

which also have the names

$8{+}10^{0}$, $8{+}10^{-1}$, $8{+}10^{-2}$, $8{+}10^{-3}$, $8{+}10^{-4}$, $8{+}10^{-5}$, $8{+}10^{-6}$, $8{+}10^{-7}$, $8{+}10^{-8}$

resulting in the following output

```
0.367879441171442
0.0000453999297624849
3.72007597602084e-44
5.07595889754946e-435
1.13548386531474e-4343
3.56294956530937e-43430
3.29683147808856e-434295
1.51693678089873e-4342945
6.45170969282177e-43429449
```

which is clearly going to zero. (If $6.45\cdots\times10^{-43,429,449}$ isn't "close enough" to zero, then I don't know what is!)

Therefore, we can finally conclude

$$\lim_{x\to8^-} f(x)=\infty, \qquad \text{and also} \qquad \lim_{x\to8^+} f(x)=0$$

which are two interesting answers in their own right. The final conclusion, however, must be

$$\lim_{x\to8} f(x)=d.n.e.$$

because we must say that a limit (in general) does not exist if the limit approaching from the left and the limit approaching from the right do not equal each other.

Caveat:

The exploration of that limit required a bit of work, but not too much, and it answered a rather difficult question. Thus, we can see that the technique of "numerical evaluation of limits" has a place in mathematics. Nonetheless, beware—numerical limits are horribly unreliable!

For example, the sum of the reciprocals of the first n prime numbers goes to infinity as n goes to infinity. To be precise, I mean the sum

$$\frac{1}{2}+\frac{1}{3}+\frac{1}{5}+\frac{1}{7}+\frac{1}{11}+\frac{1}{13}+\frac{1}{17}+\frac{1}{19}+\frac{1}{23}+\frac{1}{29}+\cdots$$

goes to infinity. This theorem has a nice but difficult proof and it would not be enjoyable to go through it at this time.

However, you'd never be able to detect this divergence via a computer. In fact, as n grows, that sum basically grows like $\log(\log n)$. (I'm simplifying this discussion slightly by removing a few details). For example, if your standard of "close enough" to infinity were 1000, you'd have to wait until $e^{e^{1000}}$ numbers were totaled. The concept of $e^{e^{1000}}$ is so large it is not easy for me to decide how to describe it. Even if your standard of "close enough" to infinity were 10, you'd have to wait until $e^{e^{10}}$ numbers were tabulated, and note that $e^{e^{10}}$ has 9565 digits!

Needless to say that if you were to attempt to calculate this with a `for` loop, long before that tabulation terminates the sun will have already turned into a red giant and we will all be burned to a crisp.

5.1.6. For Loops and Taylor Polynomials

You might know that sometimes, in calculus and higher level courses, you have to take many derivatives of a function—especially when computing a Taylor polynomial. Some calculus instructors teach Taylor polynomials early in the course and some teach them extremely late; a few even skip the topic. If you have not been exposed to Taylor polynomials before, sometimes called "Taylor series," then feel free to skip to the next section on Page 226. The discussion here is an excursion, not related to the rest of the chapter.

```
for i in range(0,10):
    print i,
    print "th derivative is",
    print diff(x^3-x,x,i)
```

As you can see, that calculates the first 9 derivatives of $x^3 - x$, most of which are zero. Also note that the "0th derivative" is the function itself. It should be noted, by the way, that Sage can compute the Taylor series for you directly—without needing you to compute all those derivatives—with one line of code. That will be explained on Page 173. This example is only for the purposes of discussion.

Another fun function is the Lorentz transform from special relativity:

$$L(v) = \frac{1}{\sqrt{1 - \frac{v^2}{c^2}}}$$

where v is the velocity of some object, and c is the speed of light. This formula describes how objects moving very fast will shrink in the direction of travel, become more massive, and have time slow down (or dilate) for them.

If we apply the previous tool to this problem, we would do it as follows:

```
f(v,c) = 1/sqrt(1-(v^2/c^2))
for i in range(0,10):
    print i,
    print "th derivative is",
    print diff(f,v,i)
```

which tells you that the 9th derivative is

```
893025*v/((-v^2/c^2 + 1)^(11/2)*c^10) +
13097700*v^3/((-v^2/c^2 + 1)^(13/2)*c^12) +
51081030*v^5/((-v^2/c^2 + 1)^(15/2)*c^14) +
72972900*v^7/((-v^2/c^2 + 1)^(17/2)*c^16) +
34459425*v^9/((-v^2/c^2 + 1)^(19/2)*c^18)
```

a calculation that I would not want to do by hand.

5.1.7. If You Already Know How to Program...

If you already know how to program, then you'll recognize at this point that Sage is operating like a programming language. In fact, Sage operates over the programming language Python.

- If you know Python, then you can use any Python commands you want, directly in Sage, without a problem.
- Next, if you do not know how to program in any language, do not worry about it at this time. You definitely do not need to know how to program to use Sage. Almost all tasks do not require programming, and those that do require programming usually do not require very much—just the rudiments which you'll learn about in the next few pages. This chapter has been written with you in mind, and you will be fairly functional as a programmer by the end of it (provided that you do all of the challenges.)
- If you know how to program, but not in Python, then rest assured that Python is an extremely easy language. Anyone who has knowledge of `C`, `C++`, Pascal, Java, or C# can learn Python with only[1] a meager investment of time and very little effort.

[1] My friend Martin Albrecht, a major Sage developer, once called Python "executable pseudocode."

- If you are an experienced programmer, then you can learn a lot more about Python at `http://www.diveintopython.org`
- If you are an intermediate programmer, then you might as well just read this chapter. You'll probably read it much more quickly than a true beginner, but that's fine. Over the course of this chapter, you'll learn a good deal of Python.

5.2. Writing Subroutines

Computer programs are often divided into many small bits and pieces that carry out separate tasks. Much to the frustration of students, the vocabulary differs from language to language. Sometimes these chunks of code are called "methods" or "procedures" but usually they are called "functions." Some very old programming languages called them "subprograms." Python definitely prefers the word "function."

This presents us with a major verbal obstacle in mathematical computing. In math, the word function means something like $f(x) = x^2 + 7x - 12$. How then can we use the word function to simultaneously mean $f(x)$ and mean a small chunk of code? Accordingly, in this chapter I will use the old-fashioned word "subroutine" to mean a segment of code that does some task, whereas I will use the word "function" to mean something like $f(x) = x^2 + 11x - 12$. My use of the word subroutine here is rather non-standard, and most Python experts will be surprised to see that word here. It is a bit old-fashioned and would sound to the ear like someone saying "gosh darn it" or "that's groovy." However, I think the chapter would be unreadable if I used the word function to describe both chunks of Python code and mathematical functions like $f(x)$.

Whether you call them methods, procedures, subprograms, functions, or subroutines, programming in general (and mathematical programming in particular) consists of designing these building blocks and then combining them into some larger software system.

5.2.1. An Example: Working with Coinage

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. For any readers who are not from the USA, these are coins which have value \$ 0.25, \$ 0.10, \$ 0.05, \$ 0.01. This is a simple task, so that we can focus on the Python and not on the underlying mathematics.

This is the code we will type:

```
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, as we learned about on Page 219.

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

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

and we get back, in return, the correct answer shown below.

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

Now it is important to remember that order matters. Consider the following three calls to this subroutine.

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

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.

```
That's a total of $ 0.660000000000000
That's a total of $ 0.480000000000000
That's a total of $ 0.610000000000000
```

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 )
```

Sage and Python will respond to the above lines with the following response, indicating that it understood what we were hoping to say, each time.

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

That's a very nice service for those brains which have trouble with orderings, whether they are dyslexic or perhaps just temporarily distracted. However, it does introduce another problem—you have to remember how to spell the name of each variable. As you can see below, I once left the "r" out of "quarters," writing instead "quaters." Since the computer has not seen this word before, it does not know how to interpret my remarks, and therefore provides the following error message.

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

One last thing before we continue. In SageMathCloud, below your code working with coinCount, type on a line by itself `coi` and hit tab without hitting enter. As you can see, one of the options presented to you is your own newly created subroutine! SageMathCloud considers it a legit part of the world in which you are operating at that time.

5.2.2. A Challenge: A Cash Register

A fun task might be to simulate a cash register. 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.

5.2.3. An Example: 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.

```
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,
    print "costing $", side_cost*left_right_area*0.01
    print "The front/back panels have an area of", front_back_area,
    print "costing $", side_cost*front_back_area*0.01
    print "The top panel has an area of", bottom_top_area,
    print "costing $", top_cost*bottom_top_area*0.01
    print "The bottom panel has an area of", bottom_top_area,
    print "costing $", bottom_cost*bottom_top_area*0.01
    print

    print "The total volume is", N( length*width*height )

    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 $", cost*0.01
```

FIGURE 1. Code to Compute the Materials Cost of an Aquarium

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 you'll find in Figure 1. The following commands were used, one at a time, after the subroutine was defined.

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

As you can see, the code has a few new features which you perhaps have not seen before. The symbol that looks like six apostrophes in a row is actually a triple quotation mark. Between the triple quotation marks you can put whatever words you want to explain what your subroutine does, to some person reading your code. Generally, it is better to use a few sentences rather than just one. This description is called a "docstring" and it is very important. Without a good docstring, it is unlikely that any other person would ever use your code. However, code-reuse is an enormously important strategy in both industry and academia. It would be phenomenally expensive to "reinvent the wheel" each time that a computer is needed to perform some task. By the way, when we used the "`?`" operator (see Section 1.10 on Page 47), what we were reading were the docstrings written by earlier Sage programmers.

The variables `top_cost`, `bottom_cost`, and `side_cost` are used to store the costs for each part of the aquarium. If I had instead used the actual number 0.5 at each place where the variable `top_cost` appears, then the code would have been less readable. Also, the price might change. If the price becomes 0.6 instead of 0.5, then maybe I might change it only in a few places—failing to change it everywhere. Then the calculations would be ruined. However, by using a variable to store the value at the top of the program, I need only change it in one spot. Making code easy to update and maintain is also extremely important both in the industrial and academic worlds.

We saw variables used like this before (see Section 1.6 on Page 32), however there is some additional useful terminology. When a variable is created and used exclusively inside of a subroutine then it is called a *local variable*. However, these variables are created outside of any subroutine, and so are visible to all the subroutines; those variables are called *global variables*. In many types of programming, global variables are discouraged but in scientific or mathematical computing, they seem to be more common. It is considered good programming style to place all the global variables at the top of the program, so that they can be easily found—however, this is not strictly necessary. Python and Sage will not object if you were to break that rule of coding etiquette.

Every other line of the code in Figure 1 was either a `print` statement or a mathematical equation. Often in scientific computing, complex tasks can be carried out with these very simple tools.

By the way, this subroutine `analyzeAquariumDesign` will become the basis of an interactive webpage using Sage. Making Interactive webpages

can be great fun, and it is the topic of Chapter 6. We will explore using this particular subroutine in Section 6.4 on Page 290.

5.2.4. A Challenge: A Cylindrical Silo

This problem is a slight twist on a classic calculus problem. A farm-storage-supply company is manufacturing cheap silos. Any particular silo is a cylinder (perhaps with radius r and height h) and it is capped with a hemisphere, also of the same radius r. The goal is to produce a silo with some specified volume, but using the minimum cost of metal. The classical calculus problem is to find the silo of a specified volume and minimum surface area. However, we're going to add some realism, and say that the cylinder metal costs 25 cents per square foot, because it is easy to manufacture, and the spherical metal costs $ 1.50 per square foot, because it is harder to manufacture. We are hoping to minimize costs, which is similar to (but not identical to) minimizing surface area.

Your challenge is to write a subroutine that takes r and h as inputs. It should compute and print

- the surface area of the cylindrical part of the silo
- the surface area of the spherical part of the silo
- the cost of each of those and the total cost
- the volume of the silo

Naturally, you'll want to consult the internet to get the appropriate formulas for the volume and surface area of a cylinder or a hemisphere. Frequently, when I'm engaged in some scientific computing, I'll have to look up a few minor formulas.

Thanks to the Internet, looking stuff up today is a lot easier now than it was when I first learned how to program in 1988. Of course, I know what those formulas are and I could type them here. However, this is a more realistic exercise when I force you to look up the formulas on the internet, yourself. It is a better simulation of working in industry, when you will have remembered the concepts and procedures of mathematics, but will have long forgotten some of the minor details.

5.2.5. Combining Loops and Subroutines

Sometimes, particularly when first learning about functions, it is neat to evaluate a function at the natural numbers: 0, 1, 2, 3, 4, 5, 6, ..., up to some cutoff value.

The following subroutine will allow us to do just that. We can pick any function we want, and we can also choose how many terms we want in the sequence formed.

```
def functionToSequence( f, howmany ):
    """This subroutine takes a function as the first input, and
    then evaluates that function at the first 'howmany' natural
    numbers. For example, if howmany is 4, then the f is
```

```
     evaluated at 0, 1, 2, 3."""
    for k in range(0, howmany):
        print f(k),

    print
```

We can also define the following functions to test our code:

```
a(x) = 3*x + 5
b(x) = 2 - x
c(x) = 7 + 10*x
d(x) = 8*exp(-0.1*x)
f(x) = 5*3^x
g(x) = (1/8)*2^x
h(x) = (0.125)*2^x
```

We can now test our code with `function_to_sequence( a, 20 )` and see what happens. Likewise, you can use $b(x)$ or $c(x)$ by replacing `a` with `b` or `c`, and so forth. You will notice some interesting behavior differences between $g(x)$ and $h(x)$, which are different to Sage despite the fact that to a mathematician, $g(x)$ and $h(x)$ are the same function. Try it yourself—I won't spoil the surprise.

Something I find really neat about Sage is that you can type code like

```
function_to_sequence( 3*x, 20 )
```

and Sage will figure out what you meant. You do not need to define a separate function like $j(x) = 3x$ to work with $3x$ as a function.

5.2.6. Another Challenge: Totaling a Sequence

Now I'd like to ask you to return to the subroutine `function_to_sequence` that we just saw immediately in the previous subsection. Modify that subroutine to also tabulate the total of the printed numbers. It should display that total on a separate line, after displaying the sequence itself.

Now test your new subroutine using the function $r(x) = 1/(x + 1.0)^4$, and see what happens. A mathematician would say that you are computing the sums of the reciprocals of the fourth powers of the positive integers.

What total do you get? For large values of `howmany`, how does that compare to the decimal approximation of the number $\pi^4/90$? If you have learned about the Riemann[2] Zeta function—written $\zeta(x)$—then you will recognize that we have just computed $\zeta(4)$. The definition is

$$\zeta(x) = \sum_{j=1}^{\infty} \frac{1}{j^x}$$

[2]Named for Georg Friedrich Bernhard Riemann (1826–1866). He is the Riemann for Riemann Geometry (the curvature of space), the Riemann Hypothesis, and he also contributed a lot to the study of prime numbers.

where j runs through the positive integers, and x might be real, complex, or imaginary.

5.3. Loops and Newton's Method

This is a long section but an extremely important one. We're going to develop a chunk of code to something mathematically useful—finding the roots of a function. As we go along, we're going to explore various ways of approaching this task. Essentially, we'll start with a reasonably good bit of code and then we'll add features, one at a time, until we have an excellent bit of code. During that process we'll see many features of programming languages, including the "if-then-else construction," "`while` loops," "optional parameters," and many others.

5.3.1. What is Newton's Method?

The next few sections of this book will explain some useful and extremely common techniques of programming. In order to provide a genuine and interesting mathematical application, we will be examining Newton's method, a famous algorithm for finding the roots of functions, or equivalently, numerical solutions to equations.

I have written this section assuming that you, the reader, have never seen Newton's method[3] before at all. If you have seen Newton's method before, such as in a calculus class, then you might want to skim this section quickly, or skip it and proceed with Section 5.3.2 on Page 235. The general concept here is that we would like to generate a subroutine that operates much like `find_root` does—and in so doing, we will learn a great deal of mathematics as well as insight into what `find_root` actually does.

Some equations can be solved with techniques of algebra. Others can be solved with creative tricks—such as trigonometric identities. However, many equations simply cannot be solved symbolically. For example, suppose I want to find one solution to

$$\sin x = 1 - x$$

There is no way to solve this equation using ordinary techniques of algebra. However, using this procedure called Newton's method, we can make a series of successive numerical approximations. These approximations are easy to calculate, and they tend to become rather good after only a few steps.

First, we observe that a solution to the above equation would be a root of the following function:

$$f(x) = 1 - x - \sin x$$

[3]As you might guess, Newton's Method is named for Isaac Newton (1642—1727).

We start with the following famous formula:

$$x_{new} = x_{old} - \frac{f(x_{old})}{f'(x_{old})}$$

that is the core of Newton's method. We will use as our initial guess $x = 0$. Then we have

$$f(x) = 1 - x - \sin x \qquad \text{and thus} \qquad f'(x) = -1 - \cos x$$

which we can use as follows:

- With $x = 0$,
  - We have $f(0) = 1$ and $f'(0) = -2$.
  - This means that $f/f' = -1/2$.
  - Thus, the next value of x will be $0 - (-1/2) = 1/2$.
- With $x = 1/2$,
  - We have $f(1/2) = 0.0205744\cdots$ and $f'(1/2) = -1.87758\cdots$.
  - Thus $f/f' = -0.0109579\cdots$.
  - The next value of x is $0.5 - (-0.0109579\cdots) = 0.510957\cdots$.
- With $x = 0.510957$,
  - Now $f(0.510957) = 0.00000307602\ldots$
  - While $f'(0.510957) = -1.87227\cdots$.
  - Accordingly, $f/f' = -0.0000164293\cdots$.
  - That means the next value of x will be

$$0.510957 - (-0.0000164293) = 0.510973\cdots .$$

- Note, if you're following along with a handheld calculator, make sure that it is set to use "radians."

If we were to discuss these results in an email or a mathematical publication, we would say that $x = 0$ is the first approximation, $x = 1/2$ is the second approximation, $x = 0.510957$ is the third approximation, and $x = 0.510973$ is the fourth approximation.

That was not a lot of effort but it wasn't a super-fast calculation either, so we might be curious how we did. We're going to plug our fourth approximation into both sides of the original equation now, and see how close we managed to get.

- The value of $\sin 0.510973 = 0.489026196\cdots$.
- The value of $1 - 0.510973 = 0.489027000\cdots$.

We achieved this phenomenal degree of accuracy with only three uses of Newton's formula. When we have the computer carryout Newton's method, we'll see that we can get accuracy to the nearest trillionth, relatively rapidly.

Actually, I rounded to the sixth significant figure at various stages, so perhaps the fourth approximation might actually have been even better if done on a computer. Using the code that we're going to develop over the next few pages, I went to the fifth approximation. The fifth approximate solution turns out to be

$$x = 0.510973429388569\cdots$$

which has the left-hand and right-hand sides of our equation coming out to equal for 15 decimal places—accuracy to the nearest quadrillionth.

Summary:

From this exercise, we can see that Newton's method has two important properties. First, once $f(x)$ and $f'(x)$ as well as the initial guess are given, then the method is very simple to carry out. Second, the method is not terribly exciting for a human being to use, because it is repetitive. After all, once the initial guess and the two functions $f(x)$ and $f'(x)$ are known, all we are doing is repeatedly using some formulas several times. These two features mean that Newton's method is ideal for a computer, and this is even more so the case when we realize that the computer has more digits of precision than we might be wiling to use with our handheld calculator.

Tabular Newton's Method:

One minor note is that if one is doing this with a handheld calculator and not a computer, then I find it much easier if one makes a little table to organize the intermediate steps. For the above example it would be:

x	$f(x)$	$f'(x)$	ratio
0	1	-2	$-1/2$
$1/2$	0.0205744	-1.87758	-0.0109579
0.510957	0.0000307602	-1.87227	-0.0000164293
0.510973			

Practice Makes Perfect:

Now, if you really want to be certain that you have this computational technique perfectly, you might want to try one example with a paper and pencil, aided by a handheld calculator. Try instead to solve

$$e^x = 10x$$

with the initial condition being $x = 4$. After you do that, you'll probably be very eager to delegate this task to a computer. Your fourth approximation, after three uses of Newton's formula, should be around $3.57715206\cdots$, with variation depending on the brand of calculator you are using and how it handles rounding internally. Indeed, we can check

$$e^{3.57715206395730} = 35.7715206395730\cdots$$

with any handheld calculator.

5.3.2. Newton's Method with a For Loop

In Figure 2 we have a good example of a first attempt in implementing Newton's Method inside of Sage. We'll now analyze that code, line by line.

```
def newton_method( f, x_old ):
     """This subroutine uses Newton's Method to find
     a root of f, and the initial condition is x_old. There
     will be 10 iterations."""
    f_prime(x) = diff( f, x )

    for j in range(0, 10):
        print "iteration=", j
        print "x_old=", x_old
        print "f(x)=", f(x_old)
        print "f'(x)=", f_prime(x_old)

        ratio = f(x_old) / f_prime(x_old)

        print "ratio=", ratio

        x_new = x_old - ratio

        print "x_new=", x_new
        print

        x_old = N(x_new)
```

FIGURE 2. Newton's Method in Sage, Version 1

The first line notifies Sage that we are about to define a subroutine. The name of it will be "`newton_method`" and it will have two parameters, called `f` and `x_old`. It is extremely important that the next few lines are indented. The degree of indentation shows which commands are subordinate to which others. Since Newton's Method requires knowledge of the derivative of the function whose root you are trying to find, we are not surprised to see the next line defining `f_prime(x)`. Note that we cannot use an apostrophe in place of the prime, as in $f'(x)$, because the apostrophe already has a meaning in Python. This minor point was first mentioned on Page 49.

Now we come to a loop that will run 10 times, as signaled by the `for` command. As you can see, the loop is mostly `print` statements, to let the user know what's going on. At this point, the syntax of the print command is likely to be familiar to you, but you can also flip back to Section 5.1.2 on Page 219.

The three lines of the loop that are not `print` statements are three assignments. The first defines "the ratio" as we saw in the previous subsection. The second assignment defines x_{new} from x_{old} according to the rules of Newton's method.

Finally, we see the line `x_old = N(x_new)` which might be confusing. Basically, during the run of the `for` loop identified by $j = 3$, the value of x_{old} will be the value that x_{new} had during the run of the `for` loop identified by $j = 2$. This command provides the link between one run of the loop and the next run of the loop.

The only surprise would be the use of the `N( )` command, which we previously had used only to convert an exact (algebraic) form of some number, perhaps hard to understand, into a floating-point form or decimal—which is easier to understand but which is necessarily only a mere approximation.

The use of `N( )` here is to force Sage to compute approximately and not exactly. That might be surprising because one of Sage's strengths is its abilities to compute exactly. However, we will see an example in the next subsection which will reveal to us why I have made this choice.

5.3.3. Testing the Code

Our examples for testing will be the following three functions.

$$\begin{aligned} a(x) &= x^5 + x + 1 \\ b(x) &= x^x - 5 \\ c(x) &= x^3 - 2x + 1 \end{aligned}$$

Take a moment to make sure that you have correctly typed these three functions, $a(x)$, $b(x)$, and $c(x)$, into Sage. For $a(x)$ we could choose an initial condition of 0.5 and so we would type

```
newton_method( a(x), 0.5 )
```

and thereby obtain an eventual answer of $x = -0.754877\cdots$, which seems reasonable. (That's the final value of x_{old}. It should be noted that $f(x_{old})$ ends with a final value of $8.32667\cdots \times 10^{-17}$, which is really very near to zero. Therefore, we can conclude that this run of our subroutine is a success. We have successfully computed the root of a quintic polynomial.

However, there's still room for improvement, as can be seen by scrolling back to Iteration 5. We see that $f(x_{old}) = -7.62168\cdots \times 10^{-14}$ there and surely this is "close enough to zero." Therefore, we have wasted computation time. That's not important when analyzing the roots of a single polynomial, as it might be a difference of a quarter second versus a half of a second. However, if our project becomes an important part in a larger program (perhaps to be executed one million times) then wasting half the computation time would be exceptionally unwise. We'll remedy this wastefulness a few pages from now.

Now we can try $b(x)$. Since $2^2 = 4$ and $3^3 = 27$, and $4 < 5 < 27$, we anticipate that the solution to $x^x = 5$ should be found between 2 and 3. Using the initial condition of 2 results in $x_{old} = 2.12937\cdots$, which seems reasonable. We can test the provided answer by asking the value of

```
2.12937248276016^2.12937248276016
```

using cut-and-paste (rather than typing). We learn the answer is

```
5.00000000000003
```

which is extremely close to 5. We see again the wastefulness of our initial code because we get an excellent answer as early as Iteration 4.

If we were to instead use $x = 3$ as the initial condition, we get the same final value of x_{old}. It matches our answer from $x = 2$ to all 14 decimal places shown, which is really amazing accuracy. It is worthwhile to ponder the accuracy implied by agreement to 15 significant digits.

Consider if you were computing the weight of an automobile of approximately 1 metric ton in weight. Accuracy to 15 significant figures would imply being accurate to the nearest nanogram—or one billionth the weight of a paperclip (one gram). It is also worth mentioning that we didn't hit 5 exactly. Methods such as Newton's Method will produce approximations—often excellent approximations—but not exact answers.

This comparison between the behavior of $b(x)$ with $x = 2$ and $x = 3$ is meant to indicate to you that the initial guesses merely need be "reasonable" and do not have to be all that good. However, a "crazy" initial guess, such as $x = 10$, produces a failure. In particular, we see that the final value of $f(x_{old}) = 1.30099 \times 10^6$ which is not nearly zero. The failure should not be shocking, however, because we are trying to find values of x such that $x^x = 5$. I'm sure that we can agree, 10^{10} is not approximately 5.

Now that we are confident that things are working well, we can wrap up our checking with $c(x)$. With an initial condition of $x = -1$ we see that $x_{old} = -1.61803\cdots$ is the final provided answer, and $f(x_{old})$ is said to be zero exactly. This is not precisely true, but what is indicated by the long string of zeros is that $f(x_{old})$ is so close to zero that the best estimate of its value that can be represented (in floating-point form) by Sage is zero itself. If we wanted to be picky, we could use cut-and-paste to ask Sage

```
c( -1.61803398874989 )
```

which evaluates to $2.84217\cdots \times 10^{-14}$. With an initial condition of $x = 0$ we see that the final value is $x_{old} = 0.618033\cdots$. This time, the value of $f(x_{old}) = 5.55111\cdots \times 10^{-17}$. That is excellent accuracy, and we can be happy that our code works for these examples.

It is interesting to note, in this $c(x)$ case, that the initial conditions of $x = -1$ and $x = 0$ both converged to good answers, but that they converged to different answers. This frequently happens when f has multiple roots. Given certain conditions, Newton's method will converge to a root—often, but not always, the nearest root.

For fun, we can reconsider our opening problem, from Section 5.3.1 on Page 233. We wanted to solve $\sin x = 1 - x$, for x.

```
u(x) = 1 - x - sin(x)
newton_method( u, 0 )
```

We can see that in the fifth iteration we have an answer accurate to 10^{-16}, and it matches the solution we had found by hand. Likewise

```
v(x) = e^x - 10*x
newton_method( v, 4 )
```

is another example that we saw earlier.

One final note is that with these last three functions, again our computations were wasteful.

- At Iteration 5, we saw f around a trillionth for $c(x)$.
- At Iteration 5, we saw f roughly 1/9 of a quadrillionth for $u(x)$.
- At Iteration 5, we saw f said to be zero for $v(x)$.
- All of these are surely "more than sufficient accuracy" for any real-world application, so we should seek out a way to "stop early."

5.3.4. Numerical vs Exact Representations

This subsection will explore $a(x)$ in a bit more detail, and will explain why we used the `N( )` command in a new and strange way.

Let's try changing `x_old = N(x_new)` into `x_old = x_new` and see what happens. If we try the initial condition of $x = 0.5$ we see no difference. However, if we try $x = 1$, we definitely see a difference. Try it now, and see what happens—it is amusing.

Even at Iteration 5 we see the answer

$$x_{old} = \frac{-12188385459018332151653005986109291290903586}{16146170242253095711911994084139951355091803}$$

which is comical. The later iterations have even more digits shown. Of course, this is not what we had in mind, therefore we should force Sage to compute numerically. The way we would do this is to change `x_old = x_new` back into `x_old = N(x_new)`. This means at the start of Iteration 2 we are guaranteed to be working numerically.

Why did this strange phenomenon of ultra-high exactitude occur with initial condition $x = 1$ but not initial condition $x = 0.5$? The reason has to do with how Sage interprets numbers. The number 0.5 is already a decimal and thus Sage assumes that it is a mere numerical approximation. However, the number 1/2, instead of 0.5, is assumed to be exact. If we use 1/2 in place of 0.5 with `x_old = x_new` instead of `x_old = N(x_new)`, we will see a comical number of digits for x_{old} even as early as Iteration 4.

It is amusing to note that using $b(x) = x^x - 5$ and exact computations produces even more catastrophic, and humorous, consequences.

5.3.5. Working with Optional and Mandatory Parameters

Perhaps the first thing that we should correct would be the wasteful computations. I'm speaking specifically about situations where we've gotten $f(x_{old})$ down to under a billionth, yet the algorithm keeps going because we require 10 iterations. However, there will be a side-benefit to this change, which will be revealed to you in a few paragraphs.

The smart thing to do is to try to have some sort of mathematical condition whereby the subroutine can determine, on its own, that it is time to stop. However, we're going to explore a less sophisticated way of handling this first, because it is much easier to explain and implement. We are going to let the person calling our subroutine specify how many iterations are to be done. Of course, that means we need a new parameter to the subroutine, one which will tell it how many iterations to carry out.

To do this, we will carry out two small changes

- `def newton_method( f, x_old ):` becomes
  `def newton_method( f, x_old, max_iterate ):`
- `for j in range(0, 10):` becomes
  `for j in range(0, max_iterate):`

The first change establishes the third input or parameter, which is named `max_iterate`, because it is the maximum number of times that the `for` loop should iterate. The second change has the value of `max_iterate` take the place of 10 in the `for` loop—in other words, the `for` loop will iterate `max_iterate` number of times instead of precisely 10 times.

However, we can do still better. The changes we made above would change our test cases.

- `newton_method( b, 2 )` becomes `newton_method( b, 2, 10 )`
- `newton_method( b, 10 )` becomes `newton_method( b, 10, 10 )`

These changes are not difficult, and simply are providing a numerical value to the `max_iterate` parameter. However, this is not optimal for several reasons. First, a programmer who has been using our old subroutine, inside her/his own programs for a long time, would now discover her/his code to be unable to work at all. That's because the calls to our subroutine in her/his programs would have two parameters instead of three, and the subroutine now requires three parameters. As a result, that programmer's code will not operate properly, and she or he will be upset[4] with you. Second, an inexperienced new programmer should not be burdened with understanding too much about Newton's method to be able to use our subroutine. We want as wide an audience as possible to be able to use our code. It would be better if we set a "standard" value for `max_iterate` and then established the capability for other programmers to "override" our standard choice. The mechanism for this in Python is an "optional" parameter.

To implement an optional parameter, we would make this change:

- The old code `def newton_method( f, x_old ):`
- or alternatively `def newton_method( f, x_old, max_iterate ):`

[4]This issue, called "backwards compatibility" is an extremely serious one in any software project involving more than one programmer. A student who wishes to have any sort of job at all must take a moment to realize that they will not be the only one programming a computer at their new company. Therefore, they should take great care to think about this issue in depth. "Breaking" the code of other people by modifying something without permission is an outstandingly effective way to get fired.

- becomes instead `def newton_method( f, x_old, max_iterate=10 ):`

The effect of this change is that if a `max_iterate` value is given, then that value will be used. However, if no value is given, then the default value of 10 will be used instead. Make this change, and then take a moment now to test each of the following calls of our subroutine, one at a time.

```
newton_method( b, 2 )
newton_method( b, 2, max_iterate=5 )
newton_method( b, 10 )
newton_method( b, 10, max_iterate=5 )
newton_method( b, 10, max_iterate=100 )
```

As you can see in the last case, given the horrible initial guess of $x = 10$ for the function $b(x) = x^x - 5$, Newton's method did converge eventually. While you have the long output of 100 iterations on the screen, it is very interesting to see where the performance transitions from "bad" to "good." In particular, we see that Iteration 23 has $f(x_{old}) = 0.318042\cdots$, which isn't really a very good approximation of zero at all, whereas Iteration 26 has $f(x_{old})$ at roughly 19 trillionths, and Iteration 27 has $f(x_{old})$ less than one quadrillionth.

When I first learned to program in the late 1980s and early 1990s, it was common in an implementation of an iterative algorithm like Newton's method to have the user press the space bar between each iteration, to give some sort of flavor of how the algorithm is evolving. That's not how software is written in the present era, but nonetheless, it presents a good thought experiment for us. Imagine yourself having pressed the space bar to signal "next iteration" 23 times, hoping to solve the problem $x^x = 5$, and getting a value of $f(x_{old})$ around 0.318042. How frustrated would you be? Perhaps very frustrated. Most of us would have quit long before the 23 iteration, and some of us might be tempted to quit at even the fifth iteration. However, a very patient person would press the button 3 or 4 more times and get an excellent answer at the 26th or 27th iteration. We can be certain now that "where to stop" is not obvious, and some degree of patience is required.

By the way, it is worth noting that optional parameters are not common among most important programming languages. It is a feature that Python makes available you, but many languages lack.

5.3.6. Returning a Value and Nesting Subroutines

The general experience of working in scientific computing or numerical programming is writing a series of subroutines that each do something important, but which mostly accomplish their goal by calling other subroutines—sometimes those written by you, but more often subroutines written by others. In this subsection we're going to experience that by using our Newton's method subroutine to compute something that wouldn't really

be computable without something akin to Newton's method or one of its rival methods.

For the sake of example, take a look at the following code snippets:

```
13 + sin( 3*pi + 1/5 )
log( 169 ) / 2
16 - sqrt( 9 - x^2 )
```

Each of those snippets makes sense and is useable because a subroutine, namely `sin`, `log`, or `sqrt` is outputting a number that can be used with the other numbers according to the rules of arithmetic. Likewise, our `newton_method` subroutine must also output a number. It only makes sense that this number should be our very best approximation, which is the final value of x_{old} and x_{new} at the end of the subroutine's run. Note, at the end of the run, it will always be the case that $x_{old} = x_{new}$, so we should not agonize over which one to select.

To accomplish this outputting of a number, we add the line

```
return x_old
```

to the end of the subroutine. This Python command tells Sage to use the final value of x_{old} as the output of our subroutine.

This change, as well as the changes of the previous subsection, result in the code that is found in Figure 3. Those lines which have changed since Version 1 have the `# changed` annotation or `# new` along side them.

The symbol `#` is a very important one. Anything after the `#` symbol on any particular line of Python code is entirely ignored. This means that you can use it for small annotations like "`# changed`" or perhaps "`# new.`"

You can also use `#` for meatier comments, such as a single-line summary to tell the reader of the code what a few lines of code are intended to do. Often if a formula comes from some sort of paper or textbook chapter, it is a good idea to note that fact with this category of "quick" comment.

At the start of this subsection I promised you that we would be using `newton_method` to accomplish a legitimate computational task. The task that I have chosen is the computing of the self-logarithm. Given some positive real number y, we want to compute an x such that $x^x = y$. Our earlier example of $b(x) = x^x - 5$ would carry out this task for the specific case of $y = 5$, but now we're going to write code that should work for any positive real number y.

You can probably guess that instead of using $b(x) = x^x - 5$, we will use $b(x) = x^x - y$. However, this is not the only change, because we need to make an initial guess. It is easy for a human being to use reasoning to compute a reasonable initial guess, but a computer does not have the power of reason. We need a quick and dirty rule to rapidly produce an initial guess—even if the guess isn't very good. We will require an initial condition that is fairly small even for large values of y. Consider $6^6 = 46,656$, and realize that even with a 5-digit y, we have the small initial guess of $x = 6$.

```
def newton_method( f, x_old, max_iterate=10 ):    # changed
    """An implementation of Newton's Method, to find a root
    of f(x) given an initial guess for x. A default of 10
    iterations will be carried out unless the optional
    parameter max_iterate is set differently."""  # changed

    f_prime(x) = diff( f, x )

    for j in range(0, max_iterate ):    # changed
        print "iteration=", j
        print "x_old=", x_old
        print "f(x)=", f(x_old)
        print "f'(x)=", f_prime(x_old)

        ratio = f(x_old) / f_prime(x_old)

        print "ratio=", ratio

        x_new = x_old - ratio

        print "x_new=", x_new
        print

        x_old = N(x_new)

    return x_old    # new
```

FIGURE 3. Newton's Method in Sage, Version 2

In the end, my choice is to make the initial guess equal to $1 + \log_{10} y$, where $\log_{10}$ means the common logarithm of y. This is not so horrible, as the following examples illustrate

$$\begin{aligned} 1 + \log_{10} 46,656 &= 5.66890\cdots \quad \text{instead of } 6 \\ 1 + \log_{10} 27 &= 2.43136\cdots \quad \text{instead of } 3 \\ 1 + \log_{10} 4 &= 1.301029\cdots \quad \text{instead of } 2 \end{aligned}$$

I have to be honest here, and tell you that I had to experiment a few times until I found an initial-guess-selection function that I fought suitable. Last but not least, we have to decide how to use the optional parameters. I decided to set `max_iterate=100` so that I can be sure that I have a very good answer. There is a downside to using 100 iterations—the output will be extremely long, and the user will have to scroll. However, we'll learn how to mitigate this issue using verbosity control on Page 246.

```
def selfLogarithm( y ):
    """This subroutine returns x such that x^x = y, where
    y is a positive real number."""

    g(x) = x^x - y
    guess = 1 + log( y, 10 )

    answer = newton_method( g, guess, max_iterate=100 )

    # what follows is merely testing code
    test = answer^answer
    print answer, "^", answer, "=", test
    print "Desired: ", y

    return answer
```

FIGURE 4. Code to Compute the Self-Logarithm, using Newton's Method (Version 2 or better).

The final code snippet is to be found in Figure 4. Note that this code absolutely requires Version 2 or higher of our `newton_method` subroutine, because Version 1 did not have a `return` statement.

Now we can test this code by typing

```
selfLogarithm( 17 )
```

5.3.7. A Challenge: Finding Parallel Tangent Lines

The goal in this challenge is to find an x-value denoted x_p, given two functions $a(x)$ and $b(x)$, such that the tangent lines of both functions at $x = x_p$ are parallel. This should be done by calling our implementation of Newton's method. The parameters would be $a(x)$ and $b(x)$, and the output should be the value of x_p.

For an even harder challenge, have the subroutine output both tangent lines in $y = mx + b$ form. (Be warned: that is a difficult challenge!)

Here are some examples:

- For $a(x) = x^2 + 2$ and $b(x) = x^3 - x$, one answer is $x_p = 1$.
  - Observe that $a'(1) = 2$ and $b'(1) = 2$
  - We get a tangent line for $a(x)$ defined by $y = 2x + 1$.
  - We get a tangent line for $b(x)$ defined by $y = 2x - 2$.
  - For a graph of this situation, see the plot on the left in Figure 5.
- For $a(x) = x^2+2$ and $b(x) = x^3-x$, the other answer is $x_p = -1/3$.
  - Observe that $a'(-1/3) = -2/3$ and $b'(-1/3) = -2/3$
  - We get a tangent line for $a(x)$ defined by $y = (-2/3)x + 17/9$.
  - We get a tangent line for $b(x)$ defined by $y = (-2/3)x+74/999$.
  - For a graph of this one, see the plot on the right in Figure 5.

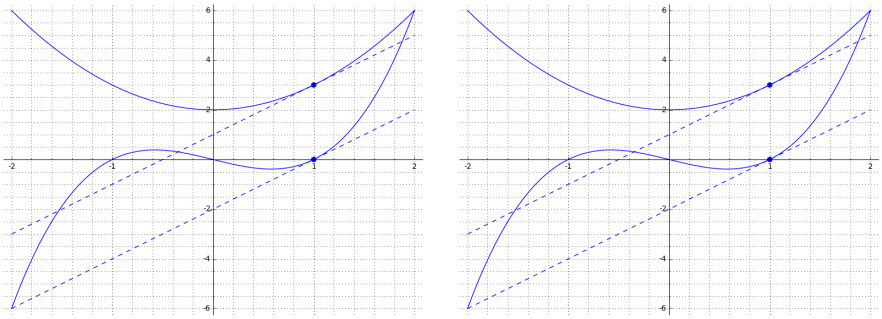


FIGURE 5. The Two Cases of Parallel Tangent Lines, from Section 5.3.7

- Your subroutine would identify only one of these two solutions, unless you did something very sophisticated.

For a different example, consider $a(x) = \sqrt{x}$ and $b(x) = x^2$. The unique solution is $x_p = \sqrt[3]{1/16}$. Observe that

$$a'(x) = b'(x) = \sqrt[3]{\frac{1}{2}} \approx 0.793700525\cdots$$

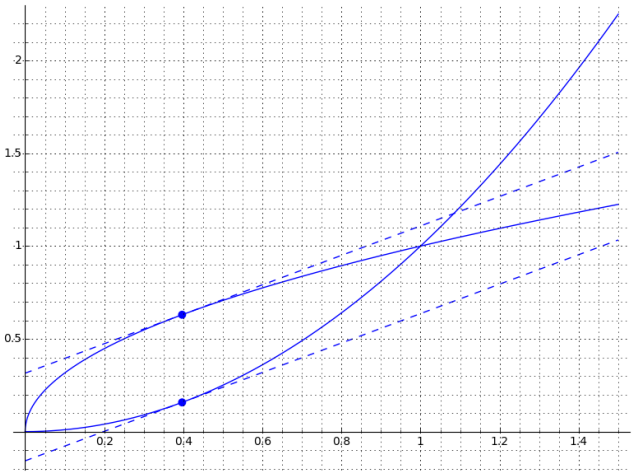
implying tangent lines of

$$y = (0.793700525\cdots)x + 0.314980262\cdots$$

for $a(x)$ and for $b(x)$

$$y = (0.793700525\cdots)x - 0.157490130\cdots$$

This last situation can be summarized by the following plot:



5.3.8. A Challenge: Halley's Method

Edmond Halley (1656–1742) was an astronomer and physicist for whom Halley's comet is named. However, he was also a friend of Isaac Newton and helped enlarge the field of study that eventually became known as calculus.

One minor discovery of Halley, which has all but been entirely forgotten, is a root-finding method that is very similar to Newton's work with Joseph Rapheson. For example, until 2007, there was no article about this method on Wikipedia, where many obscure formulas can be found. In any case, here is the underlying formula:

$$x_{new} = x_{old} - \frac{f(x_{old})f'(x_{old})}{[f'(x_{old})]^2 - \frac{1}{2}f(x_{old})f''(x_{old})}$$

As you can see, it is more complicated and it requires knowledge of the second derivative. However, it has excellent convergence whenever approaching a root r of $f(x)$, provided that both $f'(r) \neq 0$ and $f''(r) \neq 0$. Also, $f'''(x)$ must exist, therefore certain functions are excluded. In situations where $f''(x)$ and $f'(x)$ can be computed easily, Halley's method can be an extremely efficient root-finding algorithm.

Your challenge is to modify our code so that it uses Halley's method instead of Newton's method. However, I have a suggestion. First, place $f(x)$, $f'(x)$, and $f''(x)$ into "temporary variables." For example, $a = f(x_{old})$ and similarly for b and c containing the first and second derivatives. Then Halley's large formula above will be much easier to enter into the computer.

5.4. An Introduction to Control Flow

We're going to expand our capabilities in this section, while programming in Python and Sage, to include structures which control the flow of a program. The most common of these, by far, is the "if-then" construction. We'll see several examples of that, as well as how to stop a subroutine early (for example, if it has finished its computations sooner than expected), and how to announce a custom error message when things go wrong.

5.4.1. Verbosity Control

Verbosity control is when we want to limit how much output we get from one subroutine. At this moment, we are developing the `newton_method` subroutine, so we genuinely want the output—to enable us to see what is really going on. However, our Newton's method code might be part of a larger program someday. All that printing will demolish the performance of the algorithm because anything involving "I/O" (input or output) is always much slower than computation.

The plan, therefore, typically involves aggregating all the print statements into the same spot, and "protecting" them with an `if` statement.

```
def newton_method(f, x_old, max_iterate = 10, verbose=False):
    # changed

    """An implementation of Newton's Method, to find a
    root of f(x) given an initial guess for x. A default of 10
    iterations will be carried out unless the optional
    parameter max_iterate is set to something else. Set
    verbose = True to see all the intermediate stages."""
    # changed

    f_prime(x) = diff( f, x )

    for j in range(0, max_iterate):
        ratio = f(x_old) / f_prime(x_old)  # moved
        x_new = x_old - ratio    # moved

        if (verbose==True):    # new
                  print "iteration=", j
                  print "x_old=", x_old
                  print "f(x)=", f(x_old)
                  print "f'(x)=", f_prime(x_old)
                  print "ratio=", ratio
                  print "x_new=", x_new
                  print

        x_old = N(x_new)

    return x_old
```

FIGURE 6. Newton's Method in Sage, Version 3

This is our first encounter with the `if` construct, and it is a very easy to understand use of that command. We will combine this with a new parameter called "`verbose`." Like the `max_iterate` parameter, the `verbose` parameter will be optional.

If the `verbose` parameter is set to `True`, we will print, and if it is not, we will not. The code itself is given in Figure 5.4.1. Those lines which have changed (since Version 2 given in Figure 3) have the `# changed` annotation along side them, and those which have been moved have the annotation `# moved`. New lines have the `# new` annotation.

While it looks like a large number of changes, it really is not a major difference.

- I have added the optional parameter, verbose.

- I have moved all the print statements to the same spot. It is a bit of a decision as to where to put them. They have to be late enough to ensure that `ratio` and `x_new` are actually computed. However, if I had put the print statements after `x_old = N(x_new)` then the distinction between `x_old` and `x_new` would be lost.
- I added the command `if (verbose==True):`
- All the print commands were indented one level.

The indenting is very important. For example:

- The print statements are indented very far, which shows that they are subordinate both to the `if` statement and the `for` statement. That means the `if` statement will be run during each iteration of the `for` loop, and the print statements will only be executed if the parameter verbose is `True`.
- The `x_old = N(x_new)` statement is indented less, which shows that it is subordinate to the `for` statement but not to the `if` statement—in other words, the condition of verbose being `True` or `False` is of no consequence whatsoever to this line, which will be executed either way. On the other hand, the line is subordinate to the `for` loop. Therefore, each iteration of the `for` loop will see this line computed.
- The `return x_old` statement is intended very little. This means that is not subordinate to the `for` loop nor is it subordinate to the `if`. However, it is still subordinate to the `def` which means that it is indeed part of our subroutine. It will be executed once only, not once for each cycle of the `for` loop.

You can test this code with any of the following test commands:

```
newton_method( b, 2 )
newton_method( b, 2, verbose=True )
newton_method( b, 2, max_iterate=5 )
newton_method( b, 2, max_iterate=5, verbose=True )
newton_method( b, 2, verbose=True, max_iterate=5 )
```

As you can see by the last two lines of test code, the optional parameters can be given in any order. However, in the `def` line of any subroutine, the mandatory parameters must be listed first, and the optional parameters are listed after that—one cannot mingle the two. This is a rule that Sage has inherited from Python.

We have a minor technicality to discuss now. In the `if` statement, we see two equal signs, or `==`. Up to this point, we have used one equal sign. The key is that if you are *testing* equality, then you use two equal signs, but if you are *causing* equality, then you use one equal sign. For example, if we wanted to force verbosity to be `True`, we would be typing `verbose=True` at some point in the subroutine. However, in the `if` statement, we need to check equality, not force it, so we use two equal signs with `if (verbose==True):` to make that happen.

The presence of an optional verbosity parameter is a feature of my programming style. Of course, most programmers use this trick and some point or another, but for me, almost every subroutine that I write has this option. It makes debugging much simpler, and it is a good programming habit.

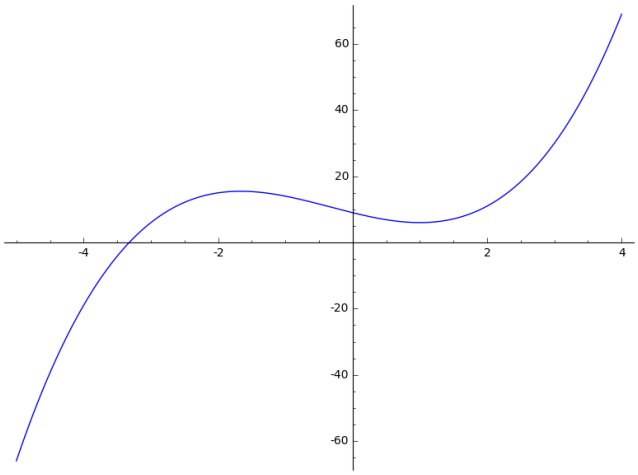
The notion of verbosity control is common in numerical computing. Sometimes this notion occurs in the humor of numerical analysts. Once at a meeting of a minor departmental committee, I was prattling on about a relatively unimportant point, and one of my colleagues got frustrated with me. He jokingly said "Greg: Enough! Verbose equals false!"

5.4.2. Theoretical Interlude: When Newton's Method Goes Crazy

Let consider the following function, which after all is just a cubic polynomial, and therefore not particularly intimidating.

$$p(x) = x^3 + x^2 - 5x + 9$$

Even its plot is not particularly unusual, as you can see below (showing $-5 < x < 4$).



However, look what we get when we try the initial condition $x = 2$ with Newton's Method and $p(x)$. The resulting splurge is an error message, and is reproduced in Figure 7. As you can see, the problem is that during Iteration 2, we see that $f'(x) = 0$.

You might think that this was one isolated example, but you'd be wrong. Here are three more examples:

- $q(x) = x^3 + x^2 - 56x + 155$ with initial condition $x = 3$.
- $r(x) = x^3 + x^2 - 85x + 323$ with initial condition $x = 3$.
- $s(x) = x^3 - x^2 - 8x + 15$ with initial condition $x = 1$.

The key problem is that Newton's Method requires us to divide by $f'(x)$ and in these examples we see that $f'(x)$ becomes zero. Since it is forbidden to divide by zero, the subroutine crashes. (The slang for "dividing by zero"

```
iteration= 0
x_old= 2
f(x)= 11
f'(x)= 11
ratio= 1
x_new= 1

iteration= 1
x_old= 1.00000000000000
f(x)= 6.00000000000000
f'(x)= 0.000000000000000

---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-1-8a48587f6c88> in <module>()
     28     return x_old    # changed
     29
---> 30 newton_method( p, Integer(2), max_iterate=Integer(20))

<ipython-input-1-8a48587f6c88> in newton_method(f, x_old, max_iterate)
     15         print "f'(x)=", f_prime(x_old)
     16
---> 17         ratio = f(x_old) / f_prime(x_old)
     18
     19         print "ratio=", ratio

/home/sageserver/sage/local/lib/python2.7/site-packages/sage/structure/el...
in sage.structure.element.RingElement.__div__ (sage/structure/element.c:1...

/home/sageserver/sage/local/lib/python2.7/site-packages/sage/symbolic/ex...
in sage.symbolic.expression.Expression._div_ (sage/symbolic/expression....

ZeroDivisionError: Symbolic division by zero
```

Note that the "..." signify parts of the error message which are not important and which had to be removed for typesetting reasons.

FIGURE 7. The Output of Newton's Method (Version 2) When it Crashes Because it Tries to Divide by Zero.

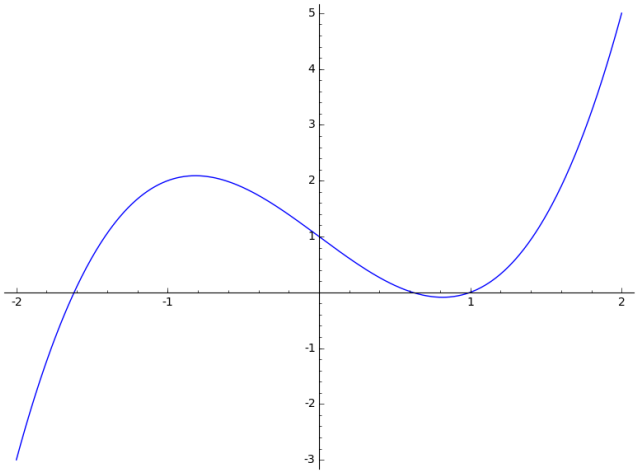
is "exploding.") The following examples are polynomials which explode on Iteration 1, and do not survive even to Iteration 2.

```
p(x) = x^2 + 8*x - 9
newton_method( p(x), -4 )

q(x) = x^4 - 2*x^2
newton_method( q(x), -1 )
newton_method( q(x), 0 )
newton_method( q(x), 1 )
```

Actually, it is relatively rare to land *exactly* upon zero. Other difficulties can occur when $f'(x)$ is approximately zero. Consider our previous example $c(x)$, again a non-intimidating cubic polynomial

$$c(x) = x^3 - 2x + 1$$

Even its plot is not particularly unusual, as you can see below (showing $-2 < x < 2$).



Therefore, we can imagine that we have a nice well-behaved function—until we type the following:

```
newton_method( c(x), -0.436485 )
```

or even better:

```
newton_method( c(x), -0.436485285968722 )
```

On my computer, the second example doesn't make progress until Iteration 79. That's because $f'(x)$ passes extremely close to zero.

The question now becomes, what are we going to do about all this? Well as it turns out, these examples represent phenomenally unlucky choices of an initial condition. If you instead select a slightly better initial condition (perhaps by graphing the function), you will see that every example in this subsection converges quickly and uneventfully.

Accordingly, what good implementations do is that if $f'(x)$ gets to be zero or near to zero, they signal an error condition so that the user can restart with a new and different choice of the initial condition. We will do this ourselves very shortly, in Section 5.5.1 on Page 255.

5.4.3. Stopping Newton's Method Early

Several times we have noticed that the computation keeps going long after a rather accurate answer has been computed. This is wasteful, particularly if our subroutine might be called millions of times inside of a larger program. For example, multivariate minimization procedures often involve an entire

Newton's method computation for each of their iterations, yet one million iterations of the multivariate solver would not be very unusual.

Accordingly, we should make some sort of cut off. If $f(x)$ is smaller than some particular value, perhaps one billionth, then we should stop the computation and report the final answer. We might take a moment to decide if we want that value to be preset or instead, set by the user. A preset value is better for users who might not be very familiar at all with Newton's method, while allowing it to be set by the user would be better for expert users. The best of both worlds can be achieved by using an optional parameter. The beginning user need not worry about (or even be told) about the optional parameter, but the expert can override when needed. Also, since the value is unlikely to be overridden too often, using an optional parameter keeps us from having too many things between the parentheses each time the subroutine is called.

The traditional name for a cut-off value of this kind is "epsilon." The Greek letter ϵ is often used to represent a value of "close enough" in computational mathematics. In Figure 5.4.3, you'll find the new code, or Version 4, of Newton's method. The key new lines are

```
if (abs(f(x_old)) < epsilon):
    return x_old
```

The first line says (in Python) what a mathematician would write as "if $|f(x)| < \epsilon$ then" and the second line, the `return` statement, tells the subroutine to return the final answer. It is implicit in that `return` statement that the subroutine has completed its computation, and that nothing more of the subroutine will be run until it is called again. Thus it is not uncommon for a subroutine to have several `return` statements, but whenever any of them are reached, the computation ends.

Some Thoughts on Absolute Values:

The way of writing absolute value in Sage is to use the `abs` command, as we first saw on Page 9. It is interesting to note why the absolute value signs are used mathematically.

Let's imagine that we instead had made the stopping condition $f(x) < \epsilon$ instead of $|f(x)| < \epsilon$. Consider $f(x) = 9 - x^2$, and suppose a confused user uses the initial condition of $x = 100$. Then we would have $f(100) = -91$. Is that a good stopping point? Is $x = 100$ an approximate root of $f(x) = 9 - x^2$? Of course not! However, $-91 < 10^{-9}$ is true. To avoid this, we need the absolute value signs to be able to write $|-91| < 10^{-9}$, which is of course false, as desired.

If you take a course in *Real Analysis* then you might see ample other examples where "if $|f(x)| < k$" is used as an abbreviation for $-k < f(x) < k$. This is a common trick in mathematical analysis.

```
def newton_method( f, x_old, max_iterate = 10,
        verbose = False, epsilon = 10^(-9) ): # changed
    """An implementation of Newton's Method, to find a root
    of f(x) given an initial guess for x. A default of 10
    iterations will be carried out unless the optional
    parameter max_iterate is set to something else. Set
    verbose = True to see all the intermediate stages.
    The algorithm will stop early if |f(x)| < epsilon."""
    # changed

    f_prime(x) = diff( f, x )

    for j in range(0, max_iterate):
        if (abs(f(x_old)) < epsilon):      # new
            return x_old      # new

        ratio = f(x_old) / f_prime(x_old)
        x_new = x_old - ratio

        if (verbose==True):
            print "iteration=", j
            print "x_old=", x_old
            print "f(x)=", f(x_old)
            print "f'(x)=", f_prime(x_old)
            print "ratio=", ratio
            print "x_new=", x_new
            print

        x_old = N(x_new)

    return x_old
```

FIGURE 8. Newton's Method in Sage, Version 4

Testing Early Termination:

We can see that this new code actually works by typing

```
b(x) = x^x - 5
newton_method( b(x), 2, verbose=True )
```

which stops very early. After Iteration 3 we have x_{new} displayed as the final output, because $f(x_{new})$ was found to be less than `epsilon`, set as its default value of one billionth.

Of course, some users might require more accuracy. In this case the code

```
b(x) = x^x - 5
newton_method( b(x), 2, epsilon=10^(-15), verbose=True )
```

demonstrates how we can achieve accuracy to a quadrillionth. Amusingly, only one additional iteration is required.

In general, it is nice to permit the user to control specific details, such as the standard of "good enough." For example, someone estimating the budget deficit of a country would probably be happy with accuracy to the nearest dollar or penny, and would not want to waste computations on getting accurate to the nearest billionth of a dollar.

5.4.4. The List of Comparators

When writing an `if` statement, there six common constructions. They are listed below in their normal mathematical sense as well as the correct Sage or Python code for them.

Mathematics	Sage or Python
if $k < 6$ then	`if (k<6):`
if $k > 6$ then	`if (k>6):`
if $k = 6$ then	`if (k==6):`
if $k \leq 6$ then	`if (k<=6):`
if $k \geq 6$ then	`if (k>=6):`
if $k \neq 6$ then	`if (k!=6):`

Note, while it looks nicer to me to write `if (k=>6)` than to write `if (k>=6)`, Python will not accept `if (k=>6)`. The correct syntax is `if (k>=6)` instead. The way that I remember this particular detail is to recall that it is common to say "is greater than or equal to" but it is uncommon to say "is equal to or greater than."

Also, we should be careful to remember to use `==` (two equal signs) instead of `=` (one equal sign) in any `if` statement, as we learned in Section 5.4.1 on Page 248.

5.4.5. A Challenge: How Many Primes are Less than x?

We've been working with Newton's method for rather a while, so I think it will be nice to consider a challenge that has nothing to do with Newton's method. Consider the following question:

For any integer $x > 1$, how many primes are found between 1 and x? More formally, how many primes p are there such that $1 \leq p \leq x$?

Here's a hint: you can make a `for` loop that counts up from 1, 2, 3, ..., $x-1$, x. Then, for each of those integers z, you could ask Sage `is_prime(z)` which will return either `True` or `False`, accordingly. Using an `if` statement, you can either increment a running counter, or not, depending if you get a `True` or `False` answer.

Now for an extra challenge (once you have completed the above) make an optional parameter `list_them` which is normally set to `False`. However,

if the user sets it to `True` then it should print all the primes that it discovers while counting up from 1 to x.

5.5. More Concepts in Flow Control

In this section, we'll explore the use of the `raise` command for unexpected situations, and more nuanced control based on something called the "if-then-else" construct.

5.5.1. Raising an "Exception"

In all sorts of programs one has to handle unusual situations that represent something going wrong. This can be a division by zero (as we discussed in Section 5.4.2 on Page 249), a negative number where one was not expected, or one of a host of other situations. The formal term for these sorts of situations is to call them *exceptions*. In the case of Newton's method, we do not want $f'(x)$ to become zero, or to approach zero.

The Python command `raise` is used to announce that an exception must be generated because some unusual condition has been detected. However, it also gives us an opportunity to make some human-readable error message, that can hopefully help the user figure out what was wrong. By the way, "raising" an exception is sometimes called "throwing" an exception in other computer languages like Java. The code that will `raise` an exception if $f'(x) = 0$ while doing Newton's Method is given in Figure 5.5.1.

For example, the code

```
c(x) = x^3 - 2*x + 1
newton_method(c(x), -0.436485285968722,verbose=True, max_iterate=20)
```

will result in the following output given in Figure 10. The hint "Please try another initial condition" in the last line of the output would cause the obedient user to type something similar to

```
c(x) = x^3 - 2*x + 1
newton_method( c(x), -0.4, verbose=True, max_iterate= 20 )
```

and then Sage cheerfully returns the correct solution, namely 0.618033.... Many other numbers could have been used in place of the -0.4 as the initial condition.

The primary change in the code to enable this feature is the line

```
raise RuntimeError('f_prime(x) got too small! '+
'Please try another initial condition.')
```

which notifies Python and Sage that an exception has occurred and that all computation should stop. It also informs Sage and Python that the exception is of the category `RuntimeError`, as well as providing a nice human-readable error message. We will not distinguish in this book between the various types of exceptions, but more advanced books on Python might.

```
def newton_method( f, x_old, max_iterate=10,
        verbose=False, epsilon=10^(-9), hazard=10^(-4) ):
        # changed

    """An implementation of Newton's Method, to find a
    root of f(x) given an initial guess for x. A default of 10
    iterations will be carried out unless the optional
    parameter max_iterate is set to something else. Set
    verbose = True to see all the intermediate stages.
    The algorithm will stop early if |f(x)| < epsilon. The
    algorithm with raise an exception if |f'(x)|<hazard."""
    # changed

    f_prime(x) = diff( f, x )

    for j in range(0, max_iterate):
        if (abs(f(x_old)) < epsilon):
            return x_old

        if (abs(f_prime(x_old)) < hazard):      # new
            raise RuntimeError('f_prime(x) got too small! '+
                    'Please try another initial condition.')
                     # new

        ratio = f(x_old) / f_prime(x_old)
        x_new = x_old - ratio

        if (verbose==True):
            print "iteration=", j
            print "x_old=", x_old
            print "f(x)=", f(x_old)
            print "f'(x)=", f_prime(x_old)
            print "ratio=", ratio
            print "x_new=", x_new
            print

        x_old = N(x_new)

    return x_old
```

FIGURE 9. Newton's Method in Sage, Version 5

```
Error in lines 30-30
Traceback (most recent call last):
  File "/projects/13762024-df4f-4043-8eca-8b21ee6f05e4/
  .sagemathcloud/sage_server.py", line 671, in execute
    exec compile(block+'\n', '', 'single') in namespace, locals
  File "", line 1, in <module>
  File "", line 14, in newton_method
RuntimeError: f_prime(x) got too small! Please try another
initial condition.
```

FIGURE 10. The Output Generated when the `raise` Command is Used.

Notice also how we used the `+` sign to make a longer error message than can fit on one line. It is not permitted to break a quotation, such as our error message, across two lines. Therefore, we wrote the quotation in two pieces—each fitting on one line—and then used the plus sign to tell Sage that it is actually one long error message. You can see the output generated in Figure 10. As always, the last line of an error message in Sage is the most important.

There are many advantages to using the `raise` command to notify the user of a major error, rather than just stopping the program. First, you could imagine that in a large program, some sequence of subroutines might call a sophisticated multivariate solver, which would call our `newton_method` subroutine. If we just return a value (such as 0) and print an error message, we are not "notifying" the higher layers that something has gone wrong, and the entire program might do very strange things as a result. However, the `raise` command would halt computation, and prevent the larger program from doing very unexpected things. Second, there is a `try: except` construct in Python, which we cannot explore here. Using that construct, you can actually respond and accommodate the error, and perhaps try the subroutine again. Third, the `raise` command allows you to offer the user a nice error message.

As you have no doubt seen, many error messages in Sage are extremely unreadable. This has no doubt caused you some frustration at times. That's the reason why you should try to avoid causing that frustration in your users and why you should make informative error messages—as well as hope that later generations of Sage developers will do similarly.

5.5.2. The If-Then-Else Construct

In Section 5.2.5 on Page 231, we wrote code that evaluates a function on the first few natural numbers, 0, 1, 2, 3, ..., for a requested number of natural numbers. However, that code will crash if we ever divide by zero. You can

try it with this function:

$$f(x) = \frac{x^2 - 4x + 3}{x^2 - 5x + 6}$$

Now we will utilize a very powerful concept in computer programming, called the "if-then-else" construct. This is like the `if` statements that we used before, but now we are adding an `else` command. The idea is that we'll first check (with an `if`) whether or not the denominator is zero. If it is zero, then we'll print "dne" for "does not exist"—that would be the "then" part. If it is not zero, then we want to print the value of the entire function at that natural number—that would be the "else" part.

Consider the following code:

```
def rationalFunctionToSequence( num, denom, howmany ):
     """This subroutine takes a function as the first input, and
     then evaluates that function at the first 'howmany' natural
     numbers. For example, if howmany is 4, then the f is
     evaluated at 0, 1, 2, 3. Any poles are reported as dne."""
    for j in range(0, howmany):
        if (denom(x=j) == 0):
            print "dne",
        else:
            value = num(x=j) / denom(x=j)
            print value,
```

Now we have two parameters for communicating the function to the subroutine, because we need the numerator and denominator separately. The third parameter tells us how many natural-number function evaluations are requested. The second line is just a `for` loop to take us through the first few natural numbers, namely 0, 1, 2, ..., $n-2$, $n-1$. Next we'll evaluate the denominator at the point $x = j$, and see if it is zero—that's the third line. In the fourth line, only to be executed if the denominator is zero, we print "dne."

The fifth line is the `else` command which means that the sixth and seventh lines, which are indented under the `else` and therefore subordinate to it, will be executed only if the denominator is not equal to zero. We compute the value of $f(x)$ at $x = j$, which is merely the numerator divided by the denominator (line six), and finally we print it in (line seven).

Now, let's test our code, with the following lines:

```
f(x) = x^2 - 4*x + 3
g(x) = x^2 - 5*x + 6
rationalFunctionToSequence( f(x), g(x), 10 )
```

We get the following output, and can be satisfied that the subroutine works correctly:

```
1/2 0 dne dne 3/2 4/3 5/4 6/5 7/6 8/7
```

The `else` command is a time-saver but it also makes the code more readable. The following code is equivalent, but more tedious to write and to read:

```
def rationalFunctionToSequence( num, denom, howmany ):
     """This subroutine takes a function as the first input, and
     then evaluates that function at the first 'howmany' natural
     numbers. For example, if howmany is 4, then the f is
     evaluated at 0, 1, 2, 3. Any poles are reported as dne."""
    for j in range(0, howmany):
        if (denom(x=j) == 0):
            print "dne",

        if (denom(x=j) != 0):
            value = num(x=j) / denom(x=j)
            print value,
```

Here, we've explicitly said that `print "dne"` is only to occur if the denominator is zero. Similarly, the division of numerator and denominator, storing it in value, and printing it, will only occur if the denominator is not zero. Therefore, the `else` command is not strictly necessary, but it does make things easier for the programmer. The advantages of having `else` available are easier to appreciate when we handle complex if-then-else statements inside each other. We won't get to that in this Chapter, but other books on Python would.

A historical note is that even though there is no "then" command in Python, nor in C, C++, nor Java, the concept is still called "if-then-else." That's because some very old programming languages, like BASIC and Pascal, used the word "then" as a command. The nomenclature reflects that historical memory.

5.5.3. Easy Challenge: Registrar's End of Semester Run, Part 1

For this challenge, you're going to write a small subroutine that will check the GPA of a student and decide their status at the university. The name of the subroutine should be `decideStatus` and there should be two parameters: first, the student's name, and second the student's GPA.

Before any processing begins, if the students GPA is less than zero or more than four, the subroutine will `raise` an exception. After that, if the student's GPA is less than 2, then they should be declared on academic probation. On a single line, print their name followed by the words "is on academic probation." On the other hand, if the students GPA is 2 or higher, then on a single line, print their name followed by "is okay."

It is important that you only use a single line, because later (in Section 5.7.9) we will make another subroutine that calls this subroutine, but for many students.

5.5.4. A Harder Challenge: End of Semester Run, Part 2

This is a modification of the previous challenge. Recall that a GPA is valid if and only if $0 \leq g \leq 4$. For any valid GPA g, the status of the student should be given according to the following rule:

$$\begin{array}{ccccc} 4 & \geq & g & \geq & 3 \quad \text{Dean's List} \\ 3 & > & g & \geq & 2 \quad \text{Normal} \\ 2 & > & g & \geq & 1 \quad \text{Academic Probation} \\ 1 & > & g & \geq & 0 \quad \text{Academic Dismissal} \end{array}$$

Here is a hint: you might want to try three if-then-else constructions. Namely, an if-then-else inside of the "then part," and another inside of the "else part," of a huge if-then-else. It requires a lot of careful indentation.

5.6. While Loops versus For Loops

As it turns out, the `for` loop is the most common loop in mathematical programming, but it has a close runner-up. The `while` loop is also very important, and occurs frequently. A `while` loop will continue as long as some logical test returns `True`; if ever the logical test returns `False`, then the loop will stop.

There is nothing more to the story, as we've now described the only difference between `for` loops versus `while` loops. Therefore, let's explore an example.

5.6.1. A Question about Factorials

Frequently when teaching about factorials, permutations, combinations, and so forth, it is useful to try to select numbers of a particular size. For simplicity, let's restrict to factorials. Let's imagine that I want to find the values of n around which $n!$ is approximately a billion, a trillion, or perhaps a thousand.

I could do this with trial and error but that's not very enjoyable. It would be more convenient to have one subroutine which just provides me with an answer. Specifically, we wish to find, given some y, the smallest value of x such that $x! \geq y$.

Since $0! = 1$, we can start with $x = 0$, and we should count up until we reach the first value of x such that $x! \geq y$. Try the following code:

```
def first_factorial_greater_than( y ):
      """This subroutine returns the lowest integer x such
      that x! >= y."""
      guess = 0
      while (factorial(guess) < y):
            guess = guess + 1

      return guess
```

This code will start with a value for `guess` that is 0. It will keep executing the command `guess = guess + 1` while it remains true that `factorial(guess) < y` evaluates `True`. However, the very moment that `factorial(guess) < y` evaluates `False`, it will halt the loop and not increment `guess`. In plainer words, the very moment that `factorial(guess) >= y` the loop stops and program flow will continue with the next line `return y`.

We can test this with a few numbers and see that it works. For example,

```
first_factorial_greater_than(130)
```

returns 6 because $5! = 120$ and $6! = 720$.

You can also try checking with the following:

```
print first_factorial_greater_than( 10^6 )
print factorial(9)
print factorial(10)
```

5.6.2. A Challenge: Finding the Next Prime Number

For this challenge, you're going to reproduce one of the commands already built into Sage. In Section 5.4.5 on Page 254, you were challenged to count the number of primes p in the interval $1 \leq p \leq x$, presumably with the use of a `for` loop and the `is_prime` command.

Here, we'd like to find the least prime greater than x. In plain English, we'd like to find the next prime after x, though there is no reason to assume x is prime. For example, the least prime greater than 13 is 17. The least prime greater than 15 is also 17.

What is interesting about this challenge is that we clearly cannot use a `for` loop because we have no idea how many integers we will have to check. For example, the next prime after 113 is 127, a jump of 14 integers. The next prime after 107 is 109, a jump of 2 integers. The next prime after 360,653 is 360,749 a jump of 96 integers.

For larger numbers, the situation is even more amusing. The next prime after 1,693,182,318,746,371 is 1,693,182,318,747,503 a jump of 1132 integers. In contrast, the next prime after that is 1,693,182,318,747,523, a jump of 20 integers.

Therefore, make a small subroutine that counts upward from a number (the input) until it reaches a prime number. It should return the first prime that it finds, which of course will greater than its input, since we only move forwards and never backwards.

You can easily test your work in this challenge, as this command is already built into Sage. The command is `next_prime`. You can see how far your skills have come, as you are now coding subroutines that are equivalent to Sage commands.

For an extra challenge, write your code so that it never bothers to check any even number greater than 2, because even numbers greater than two

obviously cannot be prime. If the input is odd, this is easy—you have to therefore detect and respond to even inputs accordingly.

5.6.3. Newton's Method with a While Loop

Now that we understand `while` loops, I have to confess that I have been a bit strange by using a `for` loop in Newton's Method. The application really calls for a `while` loop. The general plan is to repeat the iterative process until the the approximation meets the required standard by being less than epsilon (in absolute value). However, we can have an "escape route" if the maximum allowable iterations is exceeded.

The code is given in Figure 5.6.3. I have marked new lines with `# new` as before. Note that I changed the default value of `max_iterate` to 1000, to reflect the fact that this will rarely be the way that the subroutine terminates. Normally it will terminate by having the approximation be less than epsilon (in absolute value).

There is also the question of how to terminate the `while` loop after `max_iterate` iterations. As you can see, I use an `if` statement to detect the condition, then print a warning to the user. Next, I use the `break` command which we haven't seen before. The `break` command will exit out of the current loop, but resume program flow after the loop in the source code. (In other words, it continues the program at the specific point where the program would go once the loop had normally ended.) This is in contrast to a `return` command, which will terminate both the loop and the subroutine. It is a technicality worth mentioning that if you have several loops inside each other, the `break` command applies to the innermost loop only. Here, when we hit the `break` command, we will execute some `print` statements when the optional parameter `verbose` is `True`, and then we return the final answer.

5.6.4. The Impact of Repeated Roots

Last but not least, I'd like to show you a neat phenomenon in Newton's method. With the Version 6 of the above code typed in, define the following two polynomials:

```
f(x) = (x-1)^5
g(x) = (x-1)*(x-10)*(x-20)*(x+10)*(x+20)
```

Next, try

```
newton_method( f(x), 3, verbose = True )
```

but you will see that the $f'(x)$ value gets to be too small, and Newton's method aborts. We can stop this automatic shut off by setting the `hazard` optional parameter to be 10^{-9} instead of 10^{-4}, which is the default. Now, we type

```
newton_method( f(x), 3, verbose = True, hazard = 10^-9 )
```

and we see that 22 iterations were required! In contrast

```
def newton_method( f, x_old, max_iterate = 1000, # changed
        verbose=False, epsilon=10^(-9), hazard=10^(-4) ):

    """An implementation of Newton's Method, to find a root
    of f(x) given an initial guess for x. A default of 10
    iterations will be carried out unless the optional
    parameter max_iterate is set to something else. Set
    verbose = True to see all the intermediate stages.
    The algorithm will stop early if |f(x)| < epsilon. The
    algorithm with raise an exception if |f'(x)|<hazard."""

    f_prime(x) = diff( f, x )
    iterations=0

    while (abs(f(x_old)) >= epsilon):   # new
        iterations = iterations + 1     # new

        if (iterations >= max_iterate): # new
            print "Warning: Iteration Limit reached." # new
            break # new

        if (abs(f_prime(x_old)) < hazard):
            raise RuntimeError('f_prime(x) got too small! '+
                    'Please try another initial condition.')

        ratio = f(x_old) / f_prime(x_old)
        x_new = x_old - ratio

        if (verbose==True):
            print "iteration=", iterations # changed
            print "x_old=", x_old
            print "f(x)=", f(x_old)
            print "f'(x)=", f_prime(x_old)
            print "ratio=", ratio
            print "x_new=", x_new
            print
        x_old = N(x_new)

    # now the loop has concluded. (# new)
    if (verbose==True):            # new
        print iterations,          # new
        print " iterations were required." # new
        print "f(answer) = ",      # new
        print f(x_old)             # new

    return x_old
```

FIGURE 11. Newton's Method in Sage, Version 6

```
newton_method( f(x), 3, verbose = True, hazard = 10^-9 )
```

provides an answer within a mere 4 iterations. Also, it is a higher quality approximation, with $g(x_{old})$ being $-1.75\cdots\times 10^{-11}$. In contrast, $f(x_{old})$ was $6.99\cdots\times 10^{-10}$, considerably worse.

As you can see, $f(x)$ has one root with multiplicity five. Meanwhile, $g(x)$ has five roots of multiplicity one. You can choose very different polynomials, but a highly repeated root will indeed have slower convergence than a polynomial of the same degree but no repeated roots. This is a theorem that is often proved in a *Numerical Analysis* class, though not always in the first semester of that subject. That proof is too difficult for here, but experiment on polynomials with repeated roots, and you will see the phenomenon.

5.6.5. Factorization by Trial Division

We're going to continue with our theme of trying to program small bits of code that preform tasks similar to Sage capabilities. Here, we're going to try our hand at factoring, like the Sage `factor` command, but with an extremely straightforward approach—we'll just divide by prime numbers until we've factored our target number.

The target number is initially the number that we want to factor. We are going to divide our "target number" by successive primes, starting with 2. If a particular prime number divides the target, we will replace the target with the quotient, print that prime as part of the factorization, and we *will not* move to the next prime. If a particular prime number does not divide the target, we will replace the current prime number with the next prime number, as identified by Sage's `next_prime` command, but we *will not* print that prime as part of the factorization. We will stop when the target number is 1, without printing the 1.

One issue is how to decide if one number is divisible by another. Mathematicians usually use a trick called "modular arithmetic" but that's inconvenient here, because not all of my readers will know what modular arithmetic is. If you happen to know the modular arithmetic, then you know that x is divisible by y if and only if $x \mod y$ equals zero. If you do not happen to know the modular arithmetic, then just take my word that

```
if ( (target % current_prime) == 0 ):
```

as equivalent to the nonexistent but desired behavior of a command like `if ("a is divisible by b"):`.

The code for `trialDivisionFactor` might be a bit confusing. It is a mathematically non-trivial subroutine, and therefore it is worth a bit of wrestling with.

```
def trialDivisionFactor( x ):
    """Using trial division, this subroutine will print the prime
    factorization of the input x. The fact that x is an integer,
    greater than 1 is assumed."""
    print "Computing the factorization of", x, ":",
```

```
    target = x
    current_prime = 2

    while (target>1):
        if ( (target % current_prime) == 0 ):
            # target is divisible by the current prime.
            print current_prime,
            target = target / current_prime
        else:
            # target is not divisible by the current prime.
            current_prime = next_prime( current_prime )

    print
```

That `print` statement at the end of the subroutine will allow us to try multiple test cases at once. Let's try testing with

```
trialDivisionFactor( 25 )
trialDivisionFactor( 26 )
trialDivisionFactor( 27 )
```

We get the excellent output below:

```
Computing the factorization of 25 : 5 5
Computing the factorization of 26 : 2 13
Computing the factorization of 27 : 3 3 3
```

You can try that three-line test again, after removing the lonely `print` statement at the end of the subroutine. The funny output obtained will demonstrate why we need that `print` statement. A better test would be to check a bunch of consecutive integers, as below.

```
for y in range(19, 50):
    trialDivisionFactor( y )
```

If you try that check, you'll see that our code is working well.

However, the subroutine does not work well with a negative input. We'll have to fix that. For example, nothing gets printed if you try

```
trialDivisionFactor( -10 )
```

5.6.6. A Mini-Challenge: Trial Division, Stopping Early

Our subroutine is very inefficient in that if you ask it about 1,000,003 (which happens to be prime) it has to check all the primes starting with 2, 3, 5, 7, ..., 999983, 1000003, only finishing when it reaches 1000003.

The following lemma will help us. If a positive integer n is not divisible by any of the primes less than or equal to $\sqrt{n}$, then n must be prime.

Proof: Suppose n is not prime. Then consider the prime factorization of n, namely

$$n = (p_1)(p_2)(p_3)\cdots(p_\ell)$$

where $\ell \geq 2$ since n is not prime. We know that there does not exist a prime dividing n that is less than or equal to $\sqrt{n}$, so we have

$$\begin{array}{rcl} p_1 & > & \sqrt{n} \\ p_2 & > & \sqrt{n} \\ p_3 & > & \sqrt{n} \\ \vdots & \vdots & \vdots \\ p_\ell & > & \sqrt{n} \end{array}$$

Combining all these (which is only allowed because everything is positive) yields:

$$\begin{array}{rcl} (p_1)(p_2)(p_3)\cdots(p_\ell) & > & (\sqrt{n})^\ell \\ (p_1)(p_2)(p_3)\cdots(p_\ell) & > & n^{\ell/2} \\ n & > & n^{\ell/2} \end{array}$$

Since $\ell \geq 2$ then $\ell/2 \geq 1$ and $n^{\ell/2} > n^1$. We now have

$$n > n^{\ell/2} \text{ and } n^{\ell/2} > n \text{ therefore } n > n$$

which is clearly a contradiction. Therefore, our supposition was false, and n is surely prime. □

Therefore, we should check if `current_prime` is greater than the square root of `target`. If so, then we can freely halt the `while` loop because `target` is prime.

Your challenge is to modify the existing code from above to make this shortcut happen. A good test case for your work might be

$$x = 1,000,730,021$$

While you're upgrading this subroutine, we should fix the inability to factor negative numbers. Since we know that we can factor positive integers, an easy solution presents itself. Given some negative integer x, we should just factor instead $-x$, which will clearly be positive. We should put "-1" in front of the factorization to reflect the fact that the target number is negative. Lastly, if someone asks for the factorization of 0, then we should `raise` an exception. For example, if you use the built-in command in Sage, and ask `factor(0)`, then you'll get a long error message, followed by

```
ArithmeticError: Prime factorization of 0 not defined.
```

5.6.7. A Challenge: An Upgraded Cash Register

Now you're going to upgrade the code you wrote in the challenge of Section 5.2.2, simulating a cash register. Instead of telling the user that their change is $ 4.75, you should output that the change is four dollar bills, and three quarters. Software of this type is required in the kind of cash registers where the coins just pop out of the side of the register into a tray for the

customer to take. These registers, while expensive, have become popular in stores in the last decade because many people who would otherwise be working the cash registers are unable (!) to make change correctly.

It is a bit confusing for some to figure out how to do this while dispensing the fewest coins possible. That's a worthy goal, so that the machine does not have to be refilled with coins very frequently. Also, a customer would be irritated if they received 475 pennies back. Here's an abbreviated example:

If you know you have to return $3.90, just as an example, you should first give as many dollar bills as you can (in this case 3 dollar bills, as $4 is too much). Now you have to give 90 cents, and you must give as many quarters as you can (in this case, 3 quarters, because that's 75 cents which is okay, but $ 1.00 is too much). Okay, now 15 cents remain, so you should give as many dimes as you can (in this case 1, because $ 0.20 is too much). Lastly, 5 cents is left, and that's clearly one nickel. An easy way to code this is with five `while` loops (one for dollar bills, one for quarters, one for dimes, one for nickels and one for pennies) or with clever usage of the "floor" command in Sage.

5.7. How Arrays and Lists Work

We've had several encounters with the Python list concept throughout this book. The list structure in Python replaces older concepts like arrays and linked lists in other programming languages—but if you've never programmed before, don't worry about what that means for now. Here, we will learn some tricks about how to operate with lists.

Incidentally, we'll also get some exposure to how `plot` actually generates images. Special thanks to Brian Knaeble for encouraging me to include this subtopic, as 100-level students are often very curious about how graphing calculators and Sage actually produce graphs.

5.7.1. Lists of Points and Plotting

The small snippet of code is going to explore how computers generate plots. We're going to look at a simple function $f(x) = x^3 - x$, and write a subroutine that will graph it on the interval $-3/2 < x < 3/2$.

Here is the code that we will analyze.

```
f(x) = x^3 - x
list_of_points = []
verbose = False

for k in range(-150, 150):
    x = k*0.01
    y = f(x)
    new_point = (x,y)
    if (verbose==True):
```

```
        print new_point

    list_of_points = list_of_points + [new_point]

scatter_plot( list_of_points )
```

First, we define our function and second, we start with an empty list of points. (Those are our first two lines). Just as any sequence of numbers, separated by commas, and enclosed in square brackets, is a list—the empty list is simply a pair of square brackets with nothing between them. Then we quickly define a variable `verbose` which will be convenient later. Next follows the `for` loop which does most of the work.

For every iteration through the `for` loop, we compute an x-coordinate. While k will move from -150 to 150, each x-coordinate is to be very close together. In fact, we multiply by 0.01, forcing x to move from -1.5 to 1.5 in increments of 0.01. The y-coordinate is just $f(x)$. Then, we construct a point, which is an ordered pair, and it is just "x comma y." It might be interesting for you to set verbose to `True` instead of `False`, to actually see that list. Next, we update our list of points by adding together the old list, and a list consisting only of the new point. Finally, when all k values are processed and the `for` loop has completed, we make a `scatter_plot`.

The line

```
list_of_points = list_of_points + [new_point]
```

could use more explaining. In this line, we are telling Sage that the new list of points should become the old list of points, plus also the new point. The plus sign between two lists is going to result in a concatenation (or merger) of the two lists. However, unlike set theory, order matters and duplicate entries can occur. We'll explore the differences between sets and Python lists momentarily. The fact that we had to write the name of the list twice is slightly inconvenient because the name of this list is somewhat long. To alleviate this, Python actually has a shortcut

```
list_of_points.append( new_point )
```

that will stick a single element at the end of the given list.

Now the natural question would be "is this how Sage does it?" What happened here is the basic algorithm that is almost always used in tools like graphing calculators or older mathematical software. Sage actually takes a few points in tiny equally spaced intervals along the x-axis, instead of just one point. Also, those representative points within the interval are not equally spaced, but are chosen at random. We explained why on Page 3.2. Lastly, Sage will occasionally detect that more points are needed in certain subintervals because the function is "going wild there" and will evaluate extra points in those subintervals.

5.7.2. A Challenge: Making the Plot Command

Let's modify the code that we just wrote to make it more general. First, we should make it a proper subroutine with `def` and a name. Second, we should take some inputs: the function to be graphed, followed by the start point and the end point of the interval. Third, we should allow an optional parameter, which is how many points the user wants over the entire graphed interval. I think that 100 is a good default value for that optional parameter.

The tricky part is to make a formula, that given (1) the number of points desired, (2) the start of the interval, (3) the end of the interval, and (4) the loop's iterating variable, that will provide the x coordinate. That's the main difficulty of this challenge.

5.7.3. Operations on Lists

As we saw earlier, adding two lists concatenates them.

```
a = [ 2, 3, 4 ]
b = [ 1, 3 ]

print a + b
print b + a
```

We obtain the following output:

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

Here, we can really see the difference between Python lists and sets. First of all, in a mathematical set, no entry appears more than once. However, here we see that entries can appear more than once. Also, in mathematical sets, order does not matter. In Python, order matters.

Sometimes it is good to know the length of a Python list. For that, we use the `len` command. We actually saw this briefly on Page 150 when talking about determining how many divisors a number might have.

```
print len(a)
print len(b)
```

The `in` command is used with Python lists and `if` statements to make behavior dependent upon membership in a list. As you can see, 2 is a part of our list "`a`" but not a part of our list "`b`." Accordingly, the following lines will return `True` and `False` respectively, as one would expect.

```
print 2 in a
print 2 in b
```

To access specific members of a list, we use the square brackets. Computer scientists count from 0 and not from 1, for very hard to explain but very good reasons having to do with memory locations. Therefore, the first entry is entry 0, the second entry is entry 1, and third entry is entry 2, and so forth. Interestingly, there are times when you want the last entry, or the second to last entry, or the third to last entry. While I don't know of other

programming languages that do this (there might be some), Python allows you to get directly at these by asking for the "-1"th entry in the list, which is the last, or the "-2"th entry in the list, which is the second-to-last. Try the code below to demonstrate this feature.

```
C = [1, 2, 3, 4, 5, 6]
print C[0]
print C[1]
print C[2]
print
print C[-1]
print C[-2]
print C[-3]
```

We can also remove items from lists. The `remove` command does that.

```
C.remove(5)
print C
```

At times in mathematical programming, we might want to add up a list. The `sum` command will total up a list for you. Much less commonly used, but occasionally useful, is the `prod` command, which will multiply all the members of a list together. The code below correctly computes a sum of 28 and a product of 5040 for the first seven positive integers.

```
E = [ 1, 2, 3, 4, 5, 6, 7 ]
print "Sum: ", sum(E)
print "Prod: ", prod(E)
```

There is also a `shuffle` command in Python, which is great for randomizing a list. It can be useful in making games, or in scientific experiments where randomization is important for statistical reasons. I use it to make lists so that I can call on students in random order and make sure I do not call upon someone twice before I have called upon every student once.

```
roster = ["Alice", "Bob", "Charlie", "Diane", "Edward",
        "Francis", "Greg", "Harriet" ]
print roster
shuffle(roster)
print roster
shuffle(roster)
print roster
shuffle(roster)
print roster
shuffle(roster)
```

That code produced, for me, the following output, but you might get something completely different. (It is supposed to be pseudorandom output, after all.)

```
['Alice', 'Bob', 'Charlie', 'Diane', 'Edward', 'Francis', 'Greg', 'Harriet']
['Diane', 'Bob', 'Francis', 'Alice', 'Greg', 'Charlie', 'Harriet', 'Edward']
['Charlie', 'Diane', 'Greg', 'Bob', 'Harriet', 'Francis', 'Edward', 'Alice']
```

```
['Francis', 'Edward', 'Harriet', 'Greg', 'Bob', 'Diane', 'Alice', 'Charlie']
```

There are many commands to help you work with lists in Python. A great reference can be found at the URL below. It is part of a general Python-teaching website that is intended for experienced programmers. However, there's no harm in just scrolling through a webpage to see how many list-related commands there are in Python.

`http://www.diveintopython.net/native_data_types/lists.html`

I recommend some easier tutorials as "Further Reading" on Page 276, that are more suitable for beginners.

5.7.4. Looping through an Array

In this subsection we're going to explore how one can generate reports from arrays. Often I have some part of my program computing something, another part of the program generates statistics from that data, and finally a third part displays it in a useful format. Here, we'll simulate a small piece of a bit of management software. We will be writing a subroutine that generates a revenue report based on a list of names and an array of revenues.

We will take as input two lists, one of names and another of revenues. First, we have to check to make sure that they are the same length—and if not, we will `raise` an exception. Then we will step through all the names, with a `for` loop. We will print one line, giving the name and the matching revenue. Here is the code:

```
def generateRevenueReport( names, revenues ):
    """This subroutine takes in two lists, one of names, and
    another of revenues. If the lists are of unequal length, an
    exception is raised. Then a revenue report is generated,
    with each name and the revenue brought in, printed on
    one line per manager."""
    if ( len(names) != len(revenues) ):
        raise RuntimeError('The names list and the revenue'
             +'list were not the same length! Please try again.')

    for i in range(0, len(names)):
        print names[i],
        print " brought in ",
        print revenues[i]
```

Below is how we will test our code.

```
names = [ "Ned", "Ed", "Ted", "Jed", "Fred" ]
revenues = [ 200*1000, 300*1000, 400*1000, 100*1000, 600*1000 ]

generateRevenueReport( names, revenues )
```

You should receive the following as output:

```
Ned  brought in  200000
Ed  brought in  300000
Ted  brought in  400000
```

```
Jed  brought in  100000
Fred  brought in  600000
```

You're might be wondering why we checked the lengths of the lists and why we `raise` an exception if they are of different length. Let's say that I have a list with 5 items in it. When I ask for `names[0]` or `names[1]`, the number in the square brackets is called an index (plural: indices). Because I have five items, the valid list indices are 0, 1, 2, 3, 4. Also, because Python allows negative indices, I can ask for -1, -2, -3, -4, and -5. The natural numbers count from first to last, and the negative integers count from last to first. What is forbidden is to ask for indices that are greater than or equal to 5. If the length of one list is different from the length of another list, then the last index will ask for something which does not exist in the shorter list. This will generate an exception (even if we didn't use `raise`), and would halt the program.

To test the above phenomenon, add a name to the end of the names list, and call the subroutine again. The subroutine would `raise` an exception and the program stops. Now backspace over the if statement that checks the lengths, and the `raise` statement after it. See how the subroutine behaves now, when you tack on a spurious name. An error message will be generated either way. However, using an `if` followed by a `raise`, you can make the error message human readable.

Mini-Challenge: Compute the Total Revenue

I'd like to give you a mini-challenge now. Can you change the subroutine we just wrote so that it also outputs the total revenue (at the bottom of the revenue report) when finished?

You could choose to do this with an accumulator (see Section 5.1.4 on Page 221) or you could choose to do this using the `sum` command for lists (see Section 5.7.3 on Page 270). The choice is yours!

5.7.5. A Challenge: Company Profit Report

This challenge will have you write a subroutine where the three inputs are lists. The first is a list of names, followed by a list of revenues, and then a list of costs. These represent the managers of each division in a corporation, as well as their division's income and outgo. Your subroutine will generate a quarterly profit report, which is a list of names followed by the profit of each division. However, if the lists which are inputted are not of equal size, `raise` an exception.

Finally, don't forget the bottom line: the total profit should be displayed as the last line of the profit report. (That's the origin of the expression "How will this impact their bottom line?")

5.7.6. Averaging Some Numbers

In this subsection, we'll write a subroutine to average some grades, but we want to drop the lowest grade. Because this practice is not common outside the USA, I will take a moment to explain the concept.

For our non-USA readers, often college students in the USA are examined with a short written test called a "quiz" every week or every two weeks, instead of just one exam at the end of the semester (or end of the year) as is standard in the UK. In fact, when I teach, there might be 12–14 quizzes, a long final exam, and a semi-long exam in the middle of the semester called a "midterm exam." If you think about it, this is a huge difference. If a student takes *Calculus II* and *Calculus III* with me, they will end up taking about 32 different exam-like-events during the academic year. In the classical British model, they might take only two exams, or perhaps only one[5] exam.

In any case, with 13 quizzes, some faculty will "drop the lowest." This is a way of accommodating the fact that someone might be having a very bad day, or perhaps a hangover. The lowest grade gets struck from the record, and the remaining 12 will be averaged. For a compact example, imagine a class with six quizzes, where the scores were 100, 80, 90, 70, 80, and 90. In this case, the 70 is struck and the average is

$$\frac{100+80+90+80+90}{5} = \frac{440}{5} = 88$$

which is a `B+`. That is significantly better than $85.4285\cdots$, the average without dropping, which is a `B`. I've listed the code, below:

```
def averageGrades( grades ):
    """This subroutine takes a list of grades between 0 and 100,
    inclusive, as an input. Then it will compute the average of the
    grades, after dropping the lowest score."""
    total = 0
    lowest_seen = 101

    for g in grades:
        if ((g<0) or (g>100)):
            raise RuntimeError('Invalid grade in list!')

        total = total + g

        if (g < lowest_seen):
            lowest_seen = g

    numerator = total - lowest_seen
```

[5]It should be noted that some UK universities have adopted the American model. Furthermore, I know that German universities often have weekly written problem sessions in their mathematics classes too, which are basically the same concept as our weekly or biweekly quizzes.

```
    denominator = len(grades) - 1

    return N(numerator/denominator)
```

First, you might be surprised at the line:

```
for g in grades:
```

which is a different sort of syntax for the `for` loop than we've seen so far. Here, we are saying that `g` should take the value of each of the members of the list `grades` exactly once.

As you can see, we `raise` an exception if a grade is less than zero or over 100%. We've used the keyword `or` here, which means that the `if` statement will trigger if either $g < 0$ or if $g > 100$. In other situations, you can use the `and` command similarly.

We also have an accumulator, called `total`. The `lowest_seen` needs a bit of explanation. If we are in the middle of a loop, and see a grade `g` lower than `lowest_seen`, then we need to update `lowest_seen` to be this "very low" value of `g`. When the loop is finished, then surely `lowest_seen` is the lowest `g` value that we ever saw.

The only thing left is to choose a value for the start of the subroutine. I chose 101, because no grades will be over 100. Alternatively, if a grade over 100 is found, the subroutine will `raise` an exception. I am always guaranteed to find a grade in the list such that the grade is lower than 101. This way, `lowest_seen` will always be one of the grades in the list—and in fact, it will be the lowest.

You can test the above subroutine with the following output.

```
averageGrades( [ 100, 80, 90, 70, 80, 90 ] )
```

We get the score of 88 as we expected.

5.7.7. Mini-Challenge: Average both With and Without Dropping

A fun modification to the previous subroutine would be a nice challenge for you. Add an optional parameter called `merciful`. The lowest quiz should be dropped if `merciful` is `True`, but not dropped if `merciful` is `False`. The default should be `False`.

5.7.8. Mini-Challenge: Doubling Counting the Highest Quiz

Another modification to the previous mini-challenge would be to have the merciful professors count the highest quiz as double, rather than drop the lowest quiz. Try that.

5.7.9. A Challenge: The Registrar's Run, Part Three

Here is an excellent example of how small subroutines get built up into larger subroutines, and eventually programs. Consider the subroutines that you wrote in Section 5.5.3 and Section 5.5.4. Now you're asked to write a subroutine that will take a list of student names, and a list of GPAs. If the

lists are of unequal size, then `raise` an exception. If the lists are of equal size, call your previously written subroutine for each student in the list. A report should be generated that shows the student's name, their GPA, and lastly, their fate. You might have to modify your old subroutine slightly to accommodate this.

5.7.10. A Utility: Returning Only Real, Rational, or Integer Roots

Earlier, in Section 1.8 on Page 37, we learned how to use the `solve` command to find the roots of a polynomial. Naturally, this will include roots that are complex numbers. However, sometimes you only want the real roots, the rational roots, or the integer roots.

You probably have seen notation that looks like $\mathbb{C}$, $\mathbb{R}$, $\mathbb{Q}$, and $\mathbb{Z}$, to represent the complex numbers, the real numbers, the rational numbers, and the integers (respectively). To represent these, Sage has the notation `CC`, `RR`, `QQ`, and `ZZ`. We can use the `in` command as if these were Python lists, even though they are infinite[6] sets.

For example, if we are interested in

$$f(x) = x^9 - 10x^8 + 40x^7 - 100x^6 + 203x^5 - 290x^4 + 260x^3 - 200x^2 + 96x$$

we could type

```
f(x)=x^9-10*x^8+40*x^7-100*x^6+203*x^5-290*x^4+260*x^3-200*x^2+96*x
answers = solve( f(x) == 0, x )

for x in answers:
    print x
```

That would print all nine complex roots, namely

$$\{0,\ 1,\ 2,\ 3,\ 4,\ -2i,\ -i,\ i,\ 2i\}$$

but maybe we don't want to see the imaginary roots. We can try the following instead

```
f(x)=x^9-10*x^8+40*x^7-100*x^6+203*x^5-290*x^4+260*x^3-200*x^2+96*x
answers = solve( f(x) == 0, x )

for x in answers:
    if (x.rhs() in RR):
        print x
```

which will restrict us to the real roots only, excluding the four purely imaginary roots.

By the way, `rhs()` stands for "right hand side." The left hand side would be accessible by `lhs()` but would be useless in this case, since it would just be x.

---

[6] In fact, if you know about the different sizes of infinity, $\mathbb{R}$ and $\mathbb{C}$ are the "larger kind" of infinity or "uncountably infinite," sometimes called $\aleph_1$. On the other hand, $\mathbb{Q}$ and $\mathbb{Z}$ are the "smaller kind" of infinity or "countably infinite," sometimes called $\aleph_0$. The symbol $\aleph$ is the Hebrew letter called "aleph."

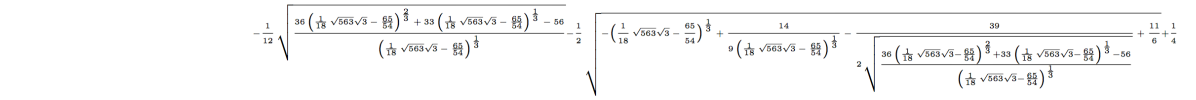


FIGURE 12. One of the Roots of $h(x)$ from Section 5.7.10

Now change $f(x)$ into $g(x)$ in the above code, where

$$g(x) = 3x^7 - 2x^6 - 15x^5 + 10x^4 + 6x^3 - 4x^2 + 24x - 16$$

taking care to remember to make the change twice—once in the definition of the polynomial and once in the `solve` command. If we keep `RR` we see only five roots. However, if we change the `RR` to `CC` then we see all seven roots, namely

$$\left\{-2,\ -\sqrt{2},\ \frac{2}{3},\ \sqrt{2},\ 2,\ -i,\ i\right\}$$

Next, we could use `QQ` and we would see the three rational roots, namely $\{-2,\ 2/3,\ 2\}$. Finally, if we use `ZZ` then we get the two integer roots, namely $\{-2,\ 2\}$.

We could replace $g(x)$ with

$$h(x) = x^4 - x^3 - x^2 - x - 1$$

and try the above code using `CC`. We get four explicit algebraic roots (with a disturbingly complicated format). One of those roots is given in Figure 12 and the others are similar in structure. If we change the `CC` to `RR`, then we discover that two of those roots are real, which means that the other two must be complex. Finally, if we change the `RR` to `QQ`, we observe that none of those roots are rational.

5.8. Where Do You Go from Here?

I hope this chapter has been useful and perhaps even enjoyable to you. Learning to program can open up whole new worlds of opportunity in research, academia, or industry. It can also sharply increase your salary.

5.8.1. Further Resources on Python Programming

Here are some resources that can allow you to grow further in your programming skills.

- A great website for people who are just starting to learn to program, and want to make that journey using Python, has been written by Alan Gauld, at the URL below. The website is a bit out of date (on July 4th, 2014 it stated that the last update was May 27th, 2007) but I do not see why that should matter.

  `http://www.freenetpages.co.uk/hp/alan.gauld/`

- Another great ebook for learning how to program, with Python as the language is *How to Think Like a Computer Scientist: Python Version* by Jeffrey Elkner, Allen B. Downey, and Chris Meyers.
  `http://openbookproject.net/thinkcs/python/english2e/`
- A highly recommended wikibook on Python for those who do not know how to program can be found at

`http://en.wikibooks.org/wiki/Non-Programmer%27s_Tutorial_for_Python`

- (I'm not in a position to judge between those three first resources.)
- A very interesting website by David Sklar and Bruce Cohen is "Just Enough Python for Numerical Analysis." This large website or small eBook is not meant to teach Python via Sage. Instead, it is focused on working with Python directly. The examples are more mathematical than the first three books above, however.
  `http://www.tetrahedra.net/m400/index.html`
- Some resources at an intermediate difficulty level are available on the official Python webpage.
  `https://www.python.org/about/gettingstarted/`
- By Mike Pilgrim, *Dive into Python!*, is available as an ebook, a website, and a physical printed book. However, it is meant for experienced programmers, who know other languages, but not Python.
  `http://www.diveintopython.net/`

5.8.2. What Has Been Left Out?

Depending on the university, we have covered roughly half of what is normally taught in a single-semester *Computer Science I*. What follows is a summary of what would be taught during the second half, for those of you who are curious.

Variable Data Types: Data in computer programming can come in various forms. Integers, floating-point numbers (which are approximations to real numbers) and strings (messages in quotes) are the most common. In most programming languages, you have to declare your variables, and name their data type. Variable declarations are not difficult or confusing, but they are important because your program won't run without them in those languages. There can be some subtle issues about choosing the right data type.

Logical Operators: We saw the use of `or` in Section 5.7.6 but a typical course would offer a few more challenging situations where the clever use of `or` as well as `and` becomes important.

Catching Exceptions that are Raised: Normally, when a `raise` command is encountered, it means some sort of catastrophic error has occurred, and the program should terminate with an error message informing the user about what went wrong. However, there is another way to use the `raise command.` Using the `try` command

is possible for a subroutine calling the offensive subroutine to remedy the situation. The program is not stopped, but instead special code is executed. Unfortunately, I cannot go into detail here.

Nested If/Then/Else Statements: The if-then-else constructor is a great way of handling two possibilities. For three or four possibilities, one can chain these together within an "else if" construct. In Python, the "`elif`" command is an abbreviation for "else if."

Handling Case/Switch Statements: For situations where there can be a dozen or more possibilities, a long stream of if-then-elif-elif-elif-else commands would be very confusing to read and debug. Most computer languages have a way handling separate "cases" and this can be useful to know about.

Scope: The brief discussion (on Page 230) of "local variables" versus "global variables" is actually a tiny taste of a moderately large issue called "scope." Understanding how scope works can be important in debugging difficult computer programs.

Recursion: In both mathematics and in computer science, recursion is when a function or subroutine calls itself. For example, $n! = (n)(n-1)!$, or $x^n = (x)(x^{n-1})$. This is not unlike a proof by induction. The examples that I've given here are very simple, but there are other, more sophisticated, examples which can be mathematically rather powerful.

Working with Strings: We worked with numerical data a great deal throughout this chapter. Likewise, one can work on string data (messages) as well. You can search strings, make substitutions, format them nicely, and do all sorts of interesting tasks. This is how word-processors work.

Assertions: These are a great way to debug some code that is not doing exactly what it should. An assertion is a logical statement, something like what you'd give to an `if` statement, but it is one that you're relatively sure should be true whenever the particular `assert` command is executed. For example, `assert (x>3)`. If it turns out that the statement is false, the program immediately stops entirely and you are notified that the assertion has failed. This is a great way to find situations where you assumed that something would always be true, but that assumption is not always true in practice.

Other Debugging Techniques: Learning to debug programs is the most vital task of all for a programmer. This consumes the vast majority of development time in practice. Most programmers develop this skill slowly over time, but it can be an enjoyable challenge. In fact, some fun exam questions can be to present a student with a task, and some code that almost carries out that task, but not exactly. The student must identify the bug and identify how to modify the code in order for it to operate correctly.

Playing Interpreter/Compiler: Another great exercise is for the student to be given code, and to be asked "what does this do?" The student must pretend to be a computer and walk through the code following the flow of control. This is also a great way to learn new and complicated algorithms.

Simulations: Generally, a simulation models some complicated engineering, physical, chemical, biological, or even sociological phenomenon. How can such a complex entity be modeled on a computer? The idea is to chop the circumstance up into many (perhaps a million) tiny slices of time. Each time-slice can then be modeled with a not-terribly-accurate model, such as a collection of linear functions. Because each time-slice is small, and there are so many of them, the simulation can still produce a high-fidelity output. This is basically the concept of an integral applied to computer science. Many intractable situations can be approached this way. The art of simulation is the easiest and yet most useful idea on this list.

Note: If a system is divided up into tiny regions of space instead of time, this is called "The Finite Element Method" and is the single "killer technique" in use today for difficult engineering situations. Many technological achievements of the last two decades have only been possible because of "The Finite Element Method." Sadly, this is almost always taught only to graduate students, and not to undergraduates, and I have no idea why.

Object-Oriented Programming: It is difficult to describe object-oriented programming in a few words. However, I will take a stab at it. At this point, you know what data is and you know what subroutines are. A collection of subroutines (called "methods") and variables containing data (called "fields") get bundled together to form a "class." For example, in a registrar's system there might be a class called "`StudentClass`" with variables for the name, the GPA, and the student-id number. Then, there would be many `StudentClass` objects, one for each student in the university. Since the data and the code are bundled together, you can do many different tasks efficiently that would otherwise be more difficult and tedious.

Naturally, for universities that use trimesters or quarters, the breakdown would be somewhat different.

More important than any of these, in an actual computer science class, you would have been given a larger battery of examples to study, and exercises to program. Both of these are phenomenally valuable and an important part of a programmer's development.

Chapter 6

Building Interactive Webpages with Sage

In this chapter, we're going to learn how to construct an interactive webpage, sometimes called an "interact" or an "app," that can be posted to the web. Unlike other uses of Sage, this particular feature is quite powerful for a simple reason: anyone with access to the internet can find your interactive webpage. Without any knowledge of Sage, or even much knowledge of mathematics, they can move some sliders around and see how a graph changes, or watch some other sort of mathematical concept unfold. Since many students are visual learners, this concept of an interactive webpage can be of phenomenal assistance to those who would otherwise be struggling to learn some chunk of mathematics.

For example, students in low-level classes (at the university or even high schools and middle schools) can be given the URL, and students can begin using the interactive webpage with very little intervention of the teacher. No experience with Sage or any other computer algebra software is needed. In fact, most students will have no idea that Sage was ever involved at all.

Personally, I had been involved with Sage for 5 years and 10 months before I learned how to make interacts over the 2012–2013 winter break. When I finally did, I was amazed at how easy it is to actually do this. Had I realized earlier how easy it is to make interactive webpages, I would certainly have begun many years previously.

Our Examples

This chapter will make reference to five diverse but elementary mathematical examples:

Tangent Line: This is our primary example. I will build this interact up slowly from scratch as we go through the main body of the chapter. The goal is to draw a moving tangent line to a simple second-degree polynomial. I will highlight how I built this applet by using my six-stage process.

Sine Wave: On the other hand, this example is my primary challenge to you. After we're done exploring the tangent-line interact, I'll invite you to do make this interact as a challenge to yourself. In case you get stuck, I've provided my own code—for each stage—so that you can refer to it.

Optimal Aquarium: Back in Section 5.2.3 on Page 228, we developed a subroutine that will compute the cost of an aquarium based upon its dimensions. I thought it might be neat to provide code that grows that subroutine into an interactive webpage. This is a relatively simple process, as it turns out.

Polynomial Grapher: This is a very quick applet meant to show you how to make pull-down menus or check boxes for your interacts. It's really quick straight-forward. You have a pull-down menu with a choice of three polynomials, and you can indicate if you'd like gridlines in your graph (or not) by using a checkbox.

Definite Integral: Previous to making the sine wave interact, I had thought of this as a good task to challenge the reader—the role now taken by the "Sine Wave" interact. As it turns out, there are some technicalities here, and it turns out that this task is a bit harder than one might first guess. I leave it as a challenge for the particularly motivated reader.

Where the Files can be Found:

The files for these five large examples can be found on my webpage. Go to `http://www.gregorybard.com/` click on "Sage Stuff" and then scroll down a bit. There is a zip file there, and one directory for each of those five interacts.

6.1. The Six-Stage Process for Building Interacts

The following multi-stage process is how I proceed to make the interacts, and this is how our explanation will be organized. In particular, we will watch how the "Tangent Line" interact was developed through each of these stages in succession.

Stage 0: Concept development—take time to actually nail down precisely what you want the interact to do. Think of this as not only a layout, but a behavior model. (If I click here, this happens; if I click there, this other thing happens, *et cetera*...)

Stage 1: Design a subroutine in Sage (using `def`) that represents one round, cycle, or phase of the interact.

Stage 2: Polish this subroutine until it is just right, with all the input and output just as you want it to be, including colors and so forth.

Stage 3: Convert this subroutine into an interact subroutine within SageMathCloud, using the "`@interact`" and "`slider`" commands.

We will describe this process in full and it really is much easier than it sounds. The `slider` command does all the work.

Stage 4: Insert this interactive subroutine into a web-page template. I have a template that[1] I have been using for a long time, and you can simply cut-and-paste your code into that one.

Stage 5: Flesh-out the webpage like any other webpage, adding some paragraphs about the mathematics, some instructions, and perhaps some motivation. You can edit this webpage like any other. Finally, upload this new webpage to your website, as you would do with any other webpage whatsoever.

The Prerequisites of Interact Building:

In writing this chapter, I have tried to aim at the broadest audience possible. Basically, there are four ingredients necessary in your background to enable you to be able to read this chapter.

(1) This chapter assumes a very basic knowledge of HTML. Only a minimal familiarity with that language, sufficient to allow the user to make their own (ordinary, non-interactive) webpages is required.
(2) It is necessary to have some background with Sage in general. For example, functionality with the contents of the majority of Chapter 1 of this book.
(3) Moreover, we will be programming in Sage via Python, and so it would be useful to have read Chapter 5 as well. However, we will only be using `def` to create some subroutines, plus we will also use some flow-of-control constructions, and so a cursory understanding of Ch. 5 will be sufficient (e.g., perhaps 5.1 and 5.2). Furthermore, it should be noted that if the reader has good experience in any programming language whatsoever (e.g. C, C++, Java, Perl, Fortran, ...), then they will be able to read the Python code and figure out what's going on without any prior knowledge of either Python or Ch. 5.
(4) Finally, the reader should be at least somewhat familiar with the SageMathCloud interface.

6.2. The Tangent-Line Interact

Now we will follow the "Tangent-Line" interact as I had built it through this six stage process.

Stage 0: Concept Development

Before actually coding anything, I try to define precisely what I want the interact to do. Generally, I have three things in mind.

[1]Provided to me by Prof. Jason Grout.

First, I want to find "the best" example possible for communicating the mathematical idea—it should not be too complex but it also should not be too simple. In this case, we want to draw a tangent line and show an early calculus student (perhaps less than three weeks into the course) what a tangent line looks like. Therefore, it is pedagogically useful to choose the drawn function to be something extremely simple—like a polynomial. I've chosen the function $f(x) = x^3 - x$.

Second, I try to focus on what the user can change, and what the user should not be permitted to change. These will become the sliders in the interact. The user should be able to change the x-coordinate of the point where the tangent line is drawn, but not the y-coordinate, which should be computed by the interact. In this way, the user will never choose a point that is not on the curve $y = x^3 - x$.

Third, I often like to make a sketch of what the output will look like. I ask myself how I can optimally display the circumstances to show sufficient detail—but not so much detail as to confuse the student. Now that my concept is finalized, I can log into SageMathCloud and begin work.

Stage 1: Design a Sage Subroutine

The code for Stage 1 of the tangent line interact can be found in Figure 1.

As you can see, first I declare the function $f(x) = x^3 - x$ and then I declare a second function for the derivative. I had Sage calculate the derivative for me, but surely I could just have defined $f'(x) = 3x^2 - 1$ instead.

The idea of having Sage compute the derivative for me is that perhaps later, I might want to use other functions, like $f(x) = \sin x$ for example. Having Sage compute the derivative for me rules out the possibility that I could forget to change the derivative from $f'(x) = 3x^2 - 1$ into $f'(x) = \cos x$, in that case.

Next, I use the `def` command to define my own subroutine which happens to be named `tangent_at_point`. This is the heart of Stage 1 of my process for making an applet. The subroutine designed at this stage is to represent the nucleus of the entire interact. It will slowly evolve into a functioning interactive webpage.

Naturally, a tangent line can be produced when I have three ingredients: the slope, the x-coordinate and the y-coordinate. The x-coordinate is going to come from a slider when the interact is completed, but now it is a real number input (called x_0) to the subroutine we are creating with the `def` command. Most interacts have multiple sliders because they need more parameters.

We have to compute the y-coordinate and the slope ourselves. Those numbers come from evaluating $f(x)$ and $f'(x)$ at x_0. Once equipped with the slope of the tangent line, we need the y-intercept. The way I teach this

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

f_prime(x) = diff( f(x), x)

def tangent_at_point( x_0 ):
    y_0 = f( x_0 )
    m = f_prime( x_0 )

    # because y - y_0 = m(x - x_0) we know that
    # y - y_0 = m*x - m*x_0
    # y = m*x + y_0 - m*x_0
    # therefore b = y_0 - m*x_0

    b = y_0 - m*x_0

    P1 = plot( f(x), -2, 2, color='blue')
    P2 = plot( m*x + b, -2, 2, color='tan')
    P3 = point( (x_0, y_0), color='red', size=100)

    P = P1+P2+P3

    P.show()

    return
```

FIGURE 1. The Code for Stage 1 of "The Tangent Line" Interact

is that I have students use the point-slope formula:

$$y - y_0 = m(x - x_0)$$

and then use algebra to convert this into $y = mx + b$ form. However, when writing a computer program, one should have an explicit formula for the y-intercept, here denoted b.

The four lines that follow are "comments" and have no effect on the computer. They are meant to tell the user what the intended effect of some nearby code *should* be, in the programmer's mind. I emphasize that comments are not infallible words of an omniscient being, but rather a statement of the programmer's intent. Any lines that have a `#` at the start are comments, and will be ignored by the computer. Here, I use that symbol to derive my shortcut formula, $b = y_0 - mx_0$.

Three `plot` commands follow. First, we want to plot the function $f(x)$ on the interval $-2 < x < 2$, and I have chosen the color blue. Second, we want to plot the tangent line $y = mx + b$ on that same interval, and I have chosen the color tan, because it sounds like the beginning of "tangent line." Next, I've added a big red point (a colored dot) at their intersection, to help

the user see where they should be looking. I chose the color red, because it usually signifies something important. Recall that superimposing plots in Sage is done by "adding" them. Therefore, I add these three plots together.

Last but not least, I have to show the superimposed plot, and finally the subroutine has concluded. This conclusion is marked with a `return` command.

If you need some reminders about adding plots, to accomplish superimposition of plots, see Page 18. For reminders about the `point` command, see Page 15.

Stage 2: Polish this Subroutine

Once you get your subroutine working, it is a good idea to tinker with it for a while and get it exactly how you want it. This is not because it is impossible to make changes later. I have even made changes after Stage 5. However, it becomes increasingly irritating to the programmer to make changes in later stages. Therefore, take time to get the colors right, the boundaries of the graph, and anything else.

In this particular case, I tested my subroutine with the following:

```
tangent_at_point(0.5)
tangent_at_point(1.5)
tangent_at_point(-1.75)
```

While the display was far from ugly, there was room for improvement. For example, in the last plot, the tangent line is relatively steep (with a slope of 8.1875), thus the tangent line actually goes up to $y = 25$ before exiting the plot. This makes the cubic curve appear tiny, because the curve goes up only to $y = 6$. Therefore, it seems to me that I must bound the y-coordinates of the plots. I did this for both $f(x)$ and the tangent line, restricting them to $-6 < y < 6$, because that is the range of $f(x) = x^3 - x$ over the domain $-2 < x < 2$.

If you look at the code in Figure 2, you can see the added `ymin` and `ymax` options inside the first two `plot` commands.

Next, I thought the red dot was a bit large, and so I cut the size from 100 down to 50. Note, this is the area of the dot and so this does not cut the diameter by half. Since the area of the dot is cut by half, the diameter is cut by $1/\sqrt{2} \approx 0.707106\cdots$.

Also, I thought a gridline background would be helpful for the plot. If you look at the command for P1, you can see that I've added the command `gridlines='minor'` to that plot command. Since it is the "bottom" of the three superimposed plots, it will show on composite superimposition as a background.

I also thought it might be nice to share with the user the numerical values of x_0, y_0, m, and the equation of the tangent line. You can see some new `print` commands after the `P.show()` statement. The exact syntax for

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

f_prime(x) = diff( f(x), x)

def tangent_at_point( x_0 ):
    y_0 = f( x_0 )
    m = f_prime( x_0 )

    # because y - y_0 = m(x - x_0) we know that
    # y - y_0 = mx - m*x_0
    # y = mx + y_0 - m*x_0
    # therefore b = y_0 - m*x_0

    b = y_0 - m*x_0

    P1 = plot( f(x), -2, 2, color='blue', ymin=-6, ymax=6,
          gridlines='minor')
    P2 = plot( m*x + b, -2, 2, color='tan', ymin=-6, ymax=6)
    P3 = point( (x_0, y_0), color='red', size=50)

    P = P1+P2+P3

    P.show()

    print
    print "x_0 = ", x_0
    print "y_0 = f(x_0) = ", y_0
    print "m = f'(x_0) = ", m
    print "tangent line: y = (", m, ")*x + (", b, ")"

    return
```

FIGURE 2. The Code for Stage 2 of "The Tangent Line" Interact

the way that the commas and quotes interact can be confusing. Don't let that be a barrier for you now—it is not important to our progress here.

Next, I reissued the $x = 0.5$, $x = 1.5$, and $x = -1.75$ test cases. They look much better now, so it is time for Stage 3.

Stage 3: Convert to an Interactive Subroutine

Now we'll learn about the `@interact` command and how to make sliders for the parameters in our subroutine. Three changes will be made in the code now, two of which are minor, but one of which takes a bit of explaining.

In particular, the first four lines

```
f_prime(x) = diff( f(x), x)

def tangent_at_point( x_0 ):
    y_0 = f( x_0 )
```

were at this time changed into

```
f_prime(x) = diff( f(x), x)

@interact
def tangent_at_point( x_0 = slider(-2, 2, 0.1, -1.5,
          label="x-coordinate") ):
    y_0 = f( x_0 )
```

First, you can see that I've added the `@interact` command, on a line by itself, above the `def` command that starts our subroutine. This command signals Sage that we are creating an interact.

Second, and this requires a bit of explaining, I have used the `slider` command to change x_0 from a numerical input into a slider that the user can manipulate. The first two parameters tell Sage that the x_0 variable will be restricted to the interval $-2 \leq x \leq 2$. The third parameter tells Sage the granularity of the slider. In this case, we want each move to represent $1/10$. Thus if the slider is at $x_0 = 0.5$, the two positions to either side would be 0.4 and 0.6. The fourth parameter represents the initial condition. In this case, the slider will start out with x_0 set to -1.5, but the user can move it around to other values. Finally, the fifth parameter (which is optional) will give a human-readable label to the slider.

Third, inside of SageMathCloud, I am no longer going to call the subroutine with commands like `tangent_at_point( -1.75 )`. Instead, I will type `tangent_at_point()`. This is because I am not passing x_0 as a number to the subroutine, but rather using a slider to change that value often.

At this point, we have an interactive subroutine inside of SageMathCloud. We should take a moment to play around with it, and see how it works. Note that when you click and drag the slider around, you must let go of the mouse button (or touchpad) before the interactive subroutine will react.

So you can see what it will look like, I've put a screen capture of the interact in Figure 3.

Stage 4: Insert into a Web Template

In this Stage, I cut-and-paste my interactive subroutine from SageMathCloud into an HTML file, thus constructing a rudimentary interactive webpage. You can use any HTML template you want for your interactive webpage; however, I have made one available to you. The template can be found as "`generictemplate.html`" in either of the project directories on SageMathCloud for this chapter. It is also given as Figure 5.

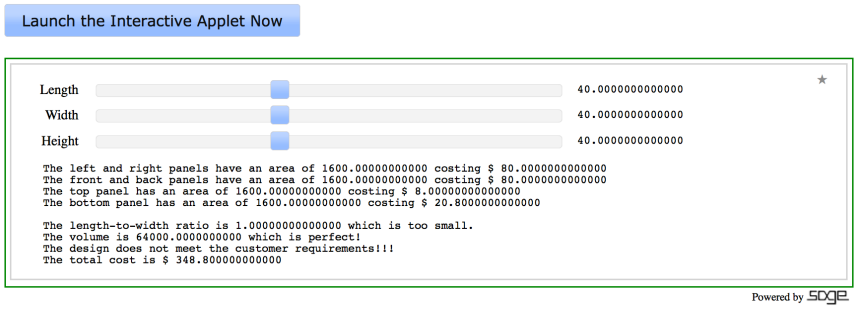


FIGURE 3. A Screen Capture of the Aquarium Interact

I'm sure that the reader knows how to cut-and-paste, but sometimes very minor things can go wrong in this process. Therefore, at the risk of seeming childish, I'm going to spell out the steps now very carefully. Please do not feel insulted.

The Exact Steps for Cutting-and-Pasting into the Template

(1) I highlighted the interactive subroutine from Stage 3 in SageMathCloud.
(2) I gave the "copy" command to my browser. (Since I use a Mac, this is "command-C" but it would be different if you are using UNIX, LINUX, or Windows.)
(3) I located the spot marked "You should cut-and-paste your Sage interactive function here" in the generic template HTML file, and highlighted it, including any lines with the # mark. (To edit HTML files, I use TextWrangler Version 3.5.3, but you can use whatever you like—even SageMathCloud—to edit HTML files.)
(4) I gave the "paste" command to my HTML editor.
(5) I saved the file using the "Save As" command and a new filename—namely, Stage4.html.
(6) I changed "YourTitleGoesHere, for the first time" into "Exploring the Tangent Line."
(7) I changed "YourTitleGoesHere, for the second time" into "Exploring the Tangent Line."
(8) I changed "YourNameGoesHere" into "Prof. Gregory V. Bard."
(9) I changed "PutTheDateHere" into "December 28th, 2013."

You can now view the HTML file in your browser. As you can see, it is very basic—only a skeleton. One can click the big blue button marked "Launch the Interactive Applet Now" and then our interact appears. Take a moment to enjoy it. We've gone from something that was just a concept to an actual interactive webpage!

Stage 5: Flesh-Out the Webpage

At this point, one can edit the webpage just like any other webpage. In the template, I have spots for "Overview," "Instructions," and "Discussion." Of course, you can have as many or as few such headings as you want. Likewise you can put as many or as few paragraphs under each one as you might like.

The final interactive web page—the output of Stage 5—can be found on my own webpage. Just go to `gregorybard.com` and click on "Interactive Web Pages for Learning Math." You'll find the interact under the title "Exploring the Tangent Line."

6.3. A Challenge to the Reader

You've now read about what needs to happen when creating an interactive webpage in Sage. I would strongly encourage you, perhaps even beg you, to take a moment now to try it yourself. There's no better way to see if your understanding is total, or to identify any minor gaps.

You can pick any task that you want, but I'd recommend the following task: Perhaps you are going to teach about sine waves as they occur in light, electricity, sound and so forth. For example, maybe you wish to explain to the student the role of amplitude and angular velocity (radians per second). In summary, imagine that you want to make an interact which will graph

$$f(t) = A\sin(\omega t)$$

where A and ω are controlled by sliders.

For the plot, I recommend using $-10 < x < 10$ and $-5 < y < 5$. For the amplitude, while one could see advantages to forcing the amplitude to be positive, I think it might be nice for students to learn the effect of $A \leq 0$, so I recommend $-5 \leq A \leq 5$ volts. For ω, it might be confusing to explain what $\omega = 0$ represents, so perhaps $0.5 \leq \omega \leq 5$ is a good choice. I had A in a granularity of each step being 1/4th of a volt and ω in a granularity of 1/2 radians per second.

Give it a try, and if you like, compare your work with my files. I recommend you try to complete all five stages before comparing your work to mine. However, if you get stuck, then feel free to take a peak at my code earlier in the process.

6.4. The Optimal Aquarium Interact

The overarching goal is to present a student with a question at the start of a course (or a unit of some course) which

(1) they will be able to understand at that time,
(2) which they cannot solve at that moment,
(3) which they might want to solve, and

(4) which they will be able to solve after completing that course (or completing that unit of the course).

The course in this case is *Calculus I* and the unit is optimization. Here, we present a problem where the dimensions for the design of an aquarium are required. The goal is to make a 64,000 cubic inch aquarium, with length-to-width ratio of 2.0, but minimum costs. The costs of the top of the aquarium are 0.5 cents per square inch, the sides are 5 cents per square inch, and the bottom is 1.3 cents per square inch. The design of the aquarium[2] must meet the two requirements of volume and length-to-width ratio (to the nearest 1%) but should do so with minimal cost.

Of course, this problem is easy to solve with calculus but the goal is to show that it is hard to solve without it, by intuition alone. One imagines a classroom full of students, most of whom won't be able to simultaneously address the volume and length-to-width ratio requirements. Of those students who have satisfied both requirements, most of them will stop at that point, ignoring the cost minimization. Then one can ask students what costs they managed to obtain, and it will become clear that the cost is not the same for all students. Next, the professor might state the optimal solution so that many of the students can see that it is indeed cheaper than their own, but perhaps not by much. Last but not least, the professor should solve the problem with explicit algebra and calculus, several lessons later, when the unit on optimization has been completed.

6.5. Selectors and Checkboxes

Below is a very simple interact, to graph a few polynomials. I wrote it quickly, to be able to show you how you can make pull-down menus or check boxes for your interacts in Sage. Here is the code:

```
@interact
def polynomial_grapher( mypoly =
         selector( [ x^2-4, x^3-x, x^2+1 ], label="Polynomial: " ),
         show_grid = False ):
    if (show_grid == True):
        P = plot( mypoly, -3, 3, ymin = -5, ymax = 5,
                gridlines="minor" )
    else:
        P = plot( mypoly, -3, 3, ymin = -5, ymax = 5 )

    show(P)
```

We see two new commands. First, `selector` will make a pull-down menu with choices being the three polynomials shown in the code. The label will

[2]For a class with middle-aged students who are "going back to school," the same problem can be posed in the guise of designing a cheaply manufactured warehouse with costs for the floor, roof, and walls.

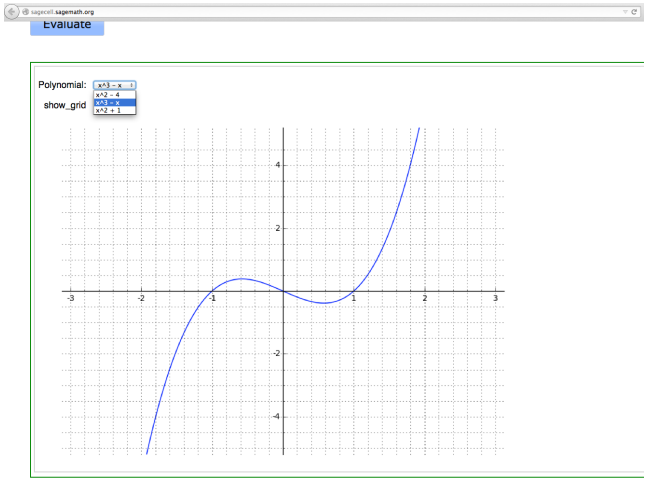


FIGURE 4. A Screen Capture of the Polynomial Grapher Applet

be the word "Polynomial:" so that the user knows the purpose of the pull-down. (I suppose in this case that was obvious, but perhaps in other cases it is not obvious.)

Second, we have an optional parameter `show_grid` that is defaulting to `False`, just as we had `verbose` default to `False` on Page 246. Whenever you have an optional parameter in a Sage interact that can be `True` or `False`, Sage knows that you want to have a box that can be checked or unchecked. The plot of our selected polynomial will be drawn with or without gridlines, depending on whether `show_grid` is `True` or `False`. The rest of the code is nothing new for us.

Figure 4 has a screen capture of this applet, but note that the checkbox has been eclipsed by the pull-down menu.

6.6. The Definite Integral Interact

At first, this interact seems really simple. We wish to explain the meaning of the definite integral, and we've chosen an easy function: $y = x^2 - 1$. The bounds of integration are $-1.5 < x < 1$ initially, but the user can change either of those bounds. The area included by the definite integral is shaded gray, to explain to the student that you shade from the curve to the x-axis. Using the "plotting integrals" technique from Section 3.1.4 on Page 95, we can make such a plot in Sage. Therefore, at first glance, this seems like it would be an easy interact to create.

As it turns out, this is not quite true. First, we have to handle the case of the student interchanging the lower and upper bounds. This will cause the

(signed) area that the applet computes to have the opposite sign, because

$$\int_a^b f(x)\,dx = -\int_b^a f(x)\,dx$$

Second, Sage reacts badly if the area to be shaded has zero area. In other words, if $a = b$, the fill-commands used by the integral plotting technique are confused, and generate an error message. Therefore, I artificially modify the width to have 10^{-12} more than it should. Therefore, the width is never zero, and the complaint is suppressed.

Finally, I decided to add a large arrow. If the bounds are in the normal order, left-to-right, then I draw a green arrow in that direction; if the bounds are "backwards," or right-to-left, then I draw a red arrow pointing right-to-left. This will help explain to students what happens if you interchange the bounds of integration.

As you can see, some of these changes are a bit unexpected. Therefore, I thought it prudent to include all the code (for all the intermediate stages) so that you can read how things progressed.

```
<!DOCTYPE HTML>
<html>
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width">
    <title>blah YourTitleGoesHere, for the first time</title>
    <script src="https://sagecell.sagemath.org/static/jquery.min.js"></script>
    <script src="https://sagecell.sagemath.org/embedded_sagecell.js"></script>
    <script>
$(function () {
    // Make *any* div with class 'compute' a Sage cell
    sagecell.makeSagecell({inputLocation: 'div.compute',
     template:       sagecell.templates.minimal,
                           evalButtonText: 'Launch the Interactive Applet Now'});
});
    </script>
  </head>
  <body style="width: 1000px;">

<! you may start modifying below this point!>
<! you may start modifying below this point!>
<! you may start modifying below this point!>

  <h1>blah YourTitleGoesHere, for the second time</h1>

<p>An Interactive Applet powered by Sage and MathJax.</p>
<p>(By YourNameGoesHere)</p>

<hr>

<h2>Overview</h2>
<p>blah blah blah</p>
<p>yada yada yada</p>
<p>blah blah blah</p>

<h2>Instructions</h2>
<p>blah blah blah</p>
<p>yada yada yada</p>

<div class="compute">
<script type="text/x-sage">

##
## You should cut-and-paste your Sage interactive function here.
##

</script>
</div>

<h2>Discussion</h2>
<p>blah blah blah</p>
<p>yada yada yada</p>
<p>blah blah blah</p>

<hr>
Last modified on PutTheDateHere.
</body>
</html>
```

FIGURE 5. The suggested HTML template for an Interactive Webpage—based on work by Prof. Jason Grout.

Appendix A

What to Do When Frustrated!

Computers are unforgiving when it comes to syntax. For users unaccustomed to that stricture, it can be very frustrating. Here I have listed for you a checklist of tips that you should use when Sage is rejecting your code for reasons that you cannot determine. I hope that these 16 questions will cover most cases.

1: Are you doing Implicit Multiplication?

This particular case is best explained with an example:

```
solve( x^3 - 5x^2 + 6x == 0, x )
```

is not correct. You must type instead

```
solve( x^3 - 5*x^2 + 6*x == 0, x )
```

In other words, Sage needs that asterisk or multiplication symbol between the coefficients 5 and 6, and the terms to which they are attached: namely x^2 and x. For some reason, trigonometric functions really cause this confusion for some users. Sage will reject

```
f(x) = 5cos( 6x )
```

but will accept

```
f(x) = 5*cos( 6*x )
```

However, for those users who find this *extremely* frustrating, there is a way out of this requirement. If you place the following line

```
implicit_multiplication(True)
```

at the start of a SageMathCloud session, then you will be in "implicit multiplication mode."

Consider the following code:

```
implicit_multiplication(True)
g=(12+3x)/(-12+2x)
plot( g, 2, 8, ymin=-30, ymax=30 )
```

At this particular moment (July 18th, 2014) this will work in SageMathCloud, but not in the Sage Cell Server. Note that the definition of g above does not have an asterisk between the 3 and the first x nor between the 2 and the second x. Because we are in "implicit multiplication mode," those two missing asterisks are forgiven.

2: Did you Forget to Declare a Variable?

The declaration of variables can be something confusing to those who have not done a lot of programming in the years prior to learning Sage. In languages such as C, C++, and Java, all the variables must be declared—no exceptions. In Sage, the variable x is pre-declared. All other variables have to be declared, except constants.

For example, if I need x, y, and z to solve a problem, but I also have a constant $g = 9.82$, then I must declare the y and the z with `var("y z")`. The g gets declared implicitly when I give the command `g=9.82`. The variable x is always pre-declared.

For stylistic reasons, some users like to declare x anyway. They would therefore type `var("x y z")`. There is no impact in Sage to the redundant declaration of x. However, visually there is a difference—it treats x in a more egalitarian way among x, y, and z.

3: Is your Internet Connection Down?

Frequently, if Sage suddenly becomes non-responsive, I realize that it is because my internet connection has become disconnected. This could be because of a wireless issue, or just the service going out for a moment or two. It might be silly to include that in this list, but it is worth mentioning. Since Sage operates through the internet, if your connection goes down, then you have to restore your connection to continue using Sage.

π: Are you Getting a Huge Error Message?

Sometimes in Sage, you'll get a huge error message, on the order of 40 lines long. This can be intimidating or disheartening even to veteran users.

The key to understanding Sage error messages is to look primarily at the last line or the last two lines. That's all you need to know on this point. However, I'll now explain *why* this is the case in the next two paragraphs, which are only intended for the curious.

What happens when you issue a command is that your code calls Sage, which probably calls something else in Sage, maybe a third call is made to part of Sage, but then flow is dispatched to some tool like Maxima, GAP, R, or Singular, that forms one of the building blocks of Sage. That hand-off might involve another nine function calls. The lines of code that were being executed when the error occurs are printed, with the deepest levels given first and the levels nearest the surface (the user interface) given last.

It is most likely that the error is in your code (the last lines printed) or possibly in Sage (the next batch of lines). The packages in Sage (e.g. Maxima, Singular, GAP, and so forth) are very well tested and debugged—those are the top two-thirds of the mess of lines. This is why you should look at the last few lines to get information to help you figure out what's wrong. Often the last line is the most informative.

4: Is the Sage Cell Server Timing Out?

Let's say that you're working on the Sage Cell Server, and you get up to go to lunch. Then you return, after about perhaps an hour and fifteen minutes. Probably, your "session" will have "timed out."

To start over without losing what you had typed into the cell, do the following:

- Highlight your code
- Give your Web Browser the "Copy" Command
- Reload the Sage Cell Server Webpage
- Give your Web Browser the "Paste" Command
- Click Evaluate

Actually, I do this so frequently, that it has become second nature. Frequently, perhaps as often as once per twenty minutes, I do these five steps as a way of making sure that my Cell Server session "stays alive." Another thought is that if the code you are working on is more than ten lines long, then perhaps you should be working in SageMathCloud, so that your work will be more permanent, and accessible to you in future months/years should you need it.

5: Are you in a Coffee House or Hotel?

The type of public wifi in a coffee house or hotel can cause issues in Sage, but they are easily remedied. First off, the routers in a hotel often cache pages for all of their users. For example, it is probable that many guests in the hotel will view the daily weather or famous news sites. Why download those for each user again and again, when they can be downloaded once and cached locally? The easiest way to do this is to make sure you're using `https` rather than `http`. With SageMathCloud this is the default. With the Sage Cell Server, you should be sure to type or bookmark:

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

Again, I emphasize the `https`. The other issue is very different. Many coffee houses require you to re-authenticate every 10 or 20 minutes. This is to make using the internet there, for long periods of time, unappealing. This way students doing academic work will not hog a seat for two hours.

Usually these re-authentications do not interact well with Sage, and so you don't actually even see a screen asking you to re-enter the wifi code. Just open a new browser window, type the URL of any large and famous

site that you haven't visited in this particular web session, and you'll see the re-authentication page. Once you've re-authenticated with whatever code is needed, you can use Sage freely again, as normal.

6: Are your Parentheses Mismatched?

In Sage, just like in mathematics, you need to have a "`(`" for each "`)`" and a "`[`" for each "`]`." This is absolutely mandatory.

Here's an old trick that was taught to me in middle school, for working with pencil-and-paper mathematics. It works very well for computers also. To see if you have screwed the parentheses up, start reading from the left of a particular line, toward the right. Start with zero. Every time you see "`(`" count upward by one, and every time you see "`)`" count downward by one. The number in your head must never become negative, and it should be zero when the line ends. If either of these conditions are violated, then something is wrong.

7: Did you Misspell a Command?

Sometimes the spelling of a command is unclear. For example, is the command actually called `A.inverse()`, `A.invert()`, or `A.find_inverse()`? Is it `find_root` or `find_roots`? Those distinctions are very hard to remember, but you have to get it correct.

The remedy to this dilemma is to first type part of the command, and then hit tab. If there is only one command that you could mean, Sage will complete the command for you. If there are several commands which would match, you get a small drop-down list. We first learned about this on Page 4.

8: Are the Parameters to a Command in the Correct Order?

Another unpleasant surprise can be when the parameters to a command are in the wrong order. For example, the `taylor` command for computing a Taylor polynomial, and the `slider` command in making applets (interactive webpages) have four parameters. Consequently, when I try to guess the order of those parameters, I often guess wrong.

The remedy to this is to type the command, on a line by itself, and to place the "`?`" operator after it. Then hit tab. This will bring up the docstring (or help file) for that command, which will have the syntax—including the parameters. We first saw this on Page 47.

9: Are the Line-Breaks not Well Placed?

There are times when Python and Sage simply insist that certain spots not contain a line break. Another issue is that the width of this printed page, in characters, is vastly smaller than a Sage Cell Server cell or the screen of SageMathCloud. Therefore, there are times when I have to break the line in this book, to make the code fit on the page, but it might not always make sense to do so on the screen.

My only word of advice is to experiment, and to be patient as you try various possibilities.

$9\frac{3}{4}$: Is there an Indentation Issue?

This is almost the same issue as the above. In Python, indentation is really important. If you're just using Sage without programming constructs like `for` loops and `while` loops, then maybe you don't have to worry about indentation. However, if you are programming in Python through Sage, as we learned in Chapter 5, then you must take care with indentation. The indentation is meaningful, as it shows which commands are subordinate to which flow-of-control constructs. Perhaps rereading Chapter 5 will help.

Sadly, when cutting-and-pasting code from the electronic version of this book into the Sage Cell Server, I often find that the indentation is destroyed or mutilated. The solution is to backspace over all the indentation, and re-indent using the tab key on your keyboard, line by line, through all of the code. This will take a lot less time than one might guess.

10: Are you Missing a Colon?

Continuing on the theme of quirks of Python that frustrate students, if you are programming in Python through Sage, as we learned in Chapter 5, then you have to remember to put a colon after commands that have some subordinate commands. These include `for` loops, `while` loops, subroutine `def`-initions, and the if-then-else construct. See Chapter 5 for examples.

11: Do you have commas in large numbers?

Large numbers in written language often have comma separators. For example, consider the following:

- The American/British notation: `1,234,567.89`
- The Continental European notation: `1.234.567,89`
- The Notation in Sage: `1234567.89`

As you can see, you must not use commas to separate the thousands, the millions, and the billions when entering numbers into Sage. That is definitely not allowed.

12: Http:// vs Https://?

I mentioned this in # 5, but I'd like to mention it again. When using the Sage Cell Server, it is much better to use `https` at the start of the URL rather than `http`. This way, your internet traffic with the server is encrypted, and intermediate routers cannot interfere with it. The danger is not so much that your mathematics will be spied upon. Instead, intermediate routers might attempt to cache the `http` traffic, an act that makes sense for nearly every type of website except mathematical software.

13: Multiple Lines of Output in the Cell Server?

Let's imagine someone teaching *College Algebra*. The instructor types

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

into the Sage Cell Server getting the usual useful result. Then, to show the students a pattern, the instructor types the following four lines into the cell.

```
factor( x^2 - 5*x + 6 )
factor( x^2 - 6*x + 8 )
factor( x^2 - 7*x + 10 )
factor( x^2 - 8*x + 12 )
```

Only the factorization of the last polynomial is given! Why is this the case? Because Sage only prints the last line of a Sage Cell computation. Instead, one should type

```
print factor( x^2 - 5*x + 6 )
print factor( x^2 - 6*x + 8 )
print factor( x^2 - 7*x + 10 )
print factor( x^2 - 8*x + 12 )
```

and then the appropriate four lines of output are given.

14: Still Not Working?

If this list of sixteen tips did not resolve your problem, then you can find useful help at the following forum. However, since it is staffed by Sage experts on an entirely volunteer basis, you might have to wait a few hours or days for a reply. It certainly is useful (and faster) to search the old postings for an answer to your question, before making a new posting of your own. Probably the question you are about to ask has been asked before. Overall, it is a very good service.

`http://ask.sagemath.org`

Appendix B

Transitioning to SageMathCloud

Chapter 1 of this book assumes you are using the Sage Cell Server. That's great for small and even some medium-sized tasks. However, you will want to switch to SageMathCloud eventually. Generally, if you are writing code of more than 20 lines, you definitely want to switch to SageMathCloud. If you are writing code of between 10–20 lines, you probably want to switch to SageMathCloud anyway. The Sage Cell Server is meant for simple tasks.

B.1. What is SageMathCloud?

Cloud computing is where dozens or hundreds of computers around the world combine to present users with one virtual supercomputer. There are many advantages of cloud computing:

Data Reliability: When you use SageMathCloud, all of your Sage worksheets will be found in your account. You will have several projects in your account, and then you can have as many Sage worksheets[1] as you like in each project. The data is stored on several machines in the cloud, scattered around the world, so there is no possible catastrophe that could destroy every existing copy simultaneously. As you can see, this is much safer than keeping all of your Sage files on one laptop.

Data Availability: All of your data is available, all the time, from any computer in the world that has an internet connection.

Load Balancing: Previously, there were many Sage notebook servers scattered around the world. However, near peak periods, it might be the case that one server is busy. In theory, if it is 4 pm in Wisconsin, it is 5 am in Beijing, and therefore you might have had better luck connecting to a machine in Beijing. However, you'd

[1] You can also organize the worksheets that belong to projects into directories and subdirectories, if you feel so inclined.

have to know where that server was (and its URL). Then all your files are still on your usual machine, perhaps in Wisconsin—you'd have to move those files yourself. All this would be a hassle. Cloud computing is the exact opposite—if one server is busy, your computational job will automatically be sent somewhere else, and you'd never even know.

Simplicity: Since all the servers are administered by the SageMathCloud community, small liberal arts colleges do not need to worry about buying a server, maintaining it, or updating the software. All the software is up-to-date all the time, and no university need take any special step, nor purchase additional equipment.

Cost Savings: There are several reasons why Cloud Computing is extremely cost effective. Normally, computing resources are not used during the hours when college students and mathematicians are asleep—3 am to 9 am. Now that all the cloud machines around the world are serving the entire world all the time, timezones no longer matter. The problem with old way of doing things—where major universities would each have their own server—is that you can't buy 0.1 servers. The lowest cost would be to buy 1 server, but perhaps a small liberal arts college only needs 0.05 servers worth of computing power. By aggregating services, this effect—called the "cost of indivisibility" in mathematical economics—is mitigated.

Collaboration: SageMathCloud has many useful collaborative features, such as making projects public/private, visible to other specific users, and online chat tools.

Checkpointing: Unless you delete a file, Sage retains not only all of your files forever, but it also takes snapshots and has all previous versions of all files. Therefore, if you screw something up, you can revert to an old version.

Finally, it is the eventual plan that around the 1% most computationally intensive users would have to pay for their computing time on SageMathCloud—but only a modest fee—and therefore the rest of the service would be made available to the typical user for free.

The only disadvantage is that you have to make an account. This takes less than a minute, but by comparison, you do not need to make an account for the Sage Cell Server.

B.2. Getting Started in SageMathCloud

The general structure of your account in SageMathCloud is that it will be divided into projects. The inter-user sharing takes place at the project level. You can have Sage worksheets in your projects, or you can divide your projects into directories and subdirectories if you prefer.

A Sage worksheet is like a sequence of giant cells from the Sage Cell Server. You can type some lines of code, and then press control-enter (which is analogous to pressing "Evaluate" in the Sage Cell Server. When you press control-enter, a horizontal line is drawn. Everything since the previous horizontal line (or the top of the sheet) is then evaluated as if it were a Sage cell on the Sage Cell Server.

Because Sage worksheets are files, you can have many of them, copy them, rename them, move them, and delete them.

This is better explained with a video. The first few minutes of this video are an overview of what Sage is about in general. After that, a demo of SageMathCloud is given. The video was made by Prof. William Stein, the founder and chief architect of Sage and the author of SageMathCloud.

`https://www.youtube.com/watch?v=6UGvrVpP89Q`

B.3. Other Cloud Features

Several other features of Cloud include using LaTeX, Python, R, and the online chat feature. We will not go into details about that here, but you can find more information at this url:

`http://cloud.sagemath.com/help`

Appendix C

Other Resources for Sage

There are many resources for Sage. Here are some major ones:

Master List of Help Resources: At the following URL you will find a registry of all the items which could go on this list. That includes tutorials, quick-start guides, reference materials, books, web tours, and videos. There is a small but growing number of resources that are written in languages other than English.
`http://www.sagemath.org/help.html`

Thematic Tutorials: This website is the housing for a very large number of Sage thematic tutorials. Those tutorials are geared at common areas of mathematics, like "Calculus," "Group Theory," "Linear Programming," or "Algebraic Combinatorics."
`http://www.sagemath.org/doc/thematic_tutorials/`

Advice Forum: This website is a place where anyone can ask a question about Sage, and an expert on Sage will answer it soon. There is vibrant activity there, since Sage is a large community.
`http://ask.sagemath.org`

Feature Tour: If you would like to show friends, parents, or non-mathematicians what Sage is about, there are two single-page feature tours. One is a bit math-heavy:
`http://www.sagemath.org/tour-quickstart.html`

The other is about the beautiful graphics that Sage can create:
`http://www.sagemath.org/tour-graphics.html`

Exploring Mathematics with Sage: This website/eBook by Paul Lutus is also about Sage, and does a good job of explaining the open-source culture of Sage development. It also has some very interesting application problems. Be sure to look at the top of your screen for the pull-down menu that shows all of the sections. Otherwise, you cannot navigate beyond the introduction.
`http://arachnoid.com/sage/index.html`

Web Tour: Aimed at mathematics faculty, the following web tour can be quite informative about some of Sage's advanced features, as well as its basic ones.
`http://www.sagemath.org/doc/a_tour_of_sage/`

Quick Reference Cards: Several Sage users have made quick reference cards, to help them learn the commands and have them handy while using Sage:
`http://wiki.sagemath.org/quickref`

The Manual: The official *Sage Reference Manual* can be found at
`http://www.sagemath.org/doc/reference/`
but this is not for beginners! Be warned, it is literally thousands of pages. This is like an encyclopedia—you don't read it cover to cover, instead you look up what you need.

Official Web Tutorial: This is the official tutorial, geared more toward graduate students, senior math majors, and faculty.
`http://www.sagemath.org/doc/tutorial/`

Web Tutorial: The following web tutorial by Mike O'Sullivan, and David Monarres has some topics we did not reach in this book. Those include abstract algebra, coding theory, and writing larger programming projects in Sage.
`http://www-rohan.sdsu.edu/~mosulliv/sagetutorial/index.html`

Numerical Tutorial: If you're doing coursework or research in numerical computing, the following tutorial is for you:
`http://www.sagemath.org/doc/numerical_sage/`

Faculty Tutorials: As the website describes itself: "This is a set of tutorials developed for the Mathematical Association of America workshops 'Sage: Using Open-Source Mathematics Software with Undergraduates' in the summers[1] of 2010-2012. The original audience for these tutorials was mathematics faculty at undergraduate institutions with little or no experience with programming or computer mathematics software. Because the computer experience required is quite minimal, this should be useful to anyone coming with little such experience to Sage."
`http://www.sagemath.org/doc/prep/index.html`

Quickstart Tutorials: These are geared toward specific undergraduate math courses, and are meant to be comprehensible to a student who has just completed that particular course. Furthermore, they are all very short. Many are a single page. They are at the same

[1]Funding provided by the National Science Foundation under grant DUE 0817071.

URL as the above tutorial, but they are listed at the bottom of the page as appendices to it.

`http://www.sagemath.org/doc/prep/index.html`

Books that Use Sage: At the time of the publication of this book, there are 24 listed books on the Sage website that use Sage for their examples. Of course, some of this are about exotic research-oriented topics meant for PhD-students in mathematics. However, several of them are entirely suitable for undergraduates. Here is the URL with the entire collection.

`http://www.sagemath.org/library-publications.html#books`

Online Video Mini-Course about Sage: A friend of mine, Travis Scrimshaw, has made a series of four videos that cover a great deal about Sage. The videos were made in July of 2013, and are therefore very up-to-date. They total 4 hours and 55 minutes of footage.

- Part I: `https://www.youtube.com/watch?v=FXbHLWtY7s4`
- Part II: `https://www.youtube.com/watch?v=9fPf1TQgFlY`
- Part III: `https://www.youtube.com/watch?v=cbBkH8Sgryw`
- Part IV: `https://www.youtube.com/watch?v=cyfv9vl6s9M`

Appendix D

Linear Systems with Infinitely-Many Solutions

This section more properly belongs in a linear algebra textbook, and not in a guide to Sage. However, I find that students often get this material wrong—during and far after the linear algebra course, and also in lower-level courses where these situations can occur. Therefore, here is a complete discussion of how to handle linear systems of equations with infinitely many solutions. They are fairly rare but not unknown in applications. This topic is not very hard, kind of deep, and good to explore.

D.1. The Opening Example

The linear system of equations given by

$$\begin{aligned} 20w + 40x + 2y + 86z &= 154 \\ 4w + 8x + 5y + 31z &= 17 \\ w + 2x + 3y + 13z &= -1 \\ -8w - 16x - 4y - 44z &= -52 \end{aligned}$$

has the matrix given below:

$$A = \begin{bmatrix} 20 & 40 & 2 & 86 & 154 \\ 4 & 8 & 5 & 31 & 17 \\ 1 & 2 & 3 & 13 & -1 \\ -8 & -16 & -4 & -44 & -52 \end{bmatrix}$$

As it turns out, the matrix A has as its RREF:

$$\texttt{A.rref() =} \begin{bmatrix} 1 & 2 & 0 & 4 & 8 \\ 0 & 0 & 1 & 3 & -3 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

We see here that there are rows of zeros, and we do not see the familiar and welcome "main diagonal" of ones. The 4×4 identity matrix is missing. Therefore, we are on alert for the possibility of an infinite number of solutions.

There is a systematic way of addressing this situation. First, we're going to identify special entries of the matrix called "pivots" or "anchors." Then, we will classify each variable as either free or determined. Third, we will rewrite the system of equations with all of the determined variables on one side of the equal signs, and all the free variables as well as constants on the opposite side. Once this is done, we will have identified each of the infinitely many solutions to this system of equations.

We're going to call the left-most non-zero entry of each row "an anchor" or "a pivot." Row 1 has as its pivot A_{11} and Row 2 has as its pivot A_{23}. Since Row 3 and Row 4 have no non-zero entries (properly, we should say because they are all-zero rows), they have no pivots. Often, when I work with pencil and paper, I will circle or underline the entries of a matrix that are pivots. In this case, I would circle or underline the entry in the top row, first column (A_{11}) and the entry in the second row, third column (A_{23}).

As we have now learned, each column in a matrix represents a variable, except the rightmost column, which represent the constants. Of course, each column either contains a pivot or it does not contain a pivot. Any column (but not the rightmost) is to be called "effective" and its variable called "determined" if the column has a pivot. Likewise, any column (but not the[1] rightmost) is to be called "defective" and its variable called "free" if the column does not have a pivot.

The rule of the previous paragraph may appear to be rather arbitrary and somewhat confusing, but let us just work with it once and see what it does for us. We will go one column at a time.

- In Column 1, we have a pivot. Therefore, Column 1 is effective and w is a determined variable.
- In Column 2, we do not have a pivot. Therefore, Column 2 is defective and x is a free variable.
- In Column 3, we have a pivot. Therefore, Column 3 is effective and y is a determined variable.
- In Column 4, we do not have a pivot. Therefore, Column 4 is defective and z is a free variable.

See, that wasn't so bad, right? Now we're going to take the RREF, translate the (non-zero) rows back into algebra, and then move the constants and free variables to the right of the equal sign, leaving only determined variables on the left of the equal sign. First, the literal or non-interpreted

[1]For the very curious: the effective/defective terminology is not used on the rightmost column because that column does not represent a variable at all. Thus it does not make sense to discuss if the variable of that rightmost column is free or determined.

translation of the RREF into algebra would be

$$\begin{aligned} w + 2x + 0y + 4z &= 8 \\ 0w + 0x + 1y + 3z &= -3 \end{aligned}$$

Now, let's move the free variables (x and z) to the right-hand side of the equal signs, to join the constants there. We obtain

$$\begin{aligned} w &= 8 - 2x - 4z \\ y &= -3 - 3z \end{aligned}$$

however, I sometimes prefer to write

$$\begin{aligned} w &= 8 - 2x - 4z \\ x &\text{ is free} \\ y &= -3 - 3z \\ z &\text{ is free} \end{aligned}$$

What the above means is that you can pick whatever values you want for x and z, but once you do, then w and y are determined by those choices that you've made.

Let's make seven specific of solutions, by choosing seven pairs of values for x and z. We obtain:

(1) If $x = 0$ and $z = 0$ then $w = 8$ and $y = -3$.
(2) If $x = 1$ and $z = 0$ then $w = 6$ and $y = -3$.
(3) If $x = 0$ and $z = 1$ then $w = 4$ and $y = -6$.
(4) If $x = 1$ and $z = 1$ then $w = 2$ and $y = -6$.
(5) If $x = 2$ and $z = 3$ then $w = -8$ and $y = -12$.
(6) If $x = -4$ and $z = -6$ then $w = 40$ and $y = 15$.
(7) If $x = \pi$ and $z = \sqrt{2}$ then $w = 8 - 2\pi - 4\sqrt{2}$ and $y = -3 - 3\sqrt{2}$.

Take a moment to see that each of these variable assignments in fact does satisfy each of the given equations. Do this by hand for the first five. We'll use Sage to verify the last two. For the sixth entry in our list, we type

```
w=40
x=-4
y=15
z=-6
print 20*w + 40*x + 2*y + 86*z
print 4*w + 8*x + 5*y + 31*z
print w + 2*x + 3*y + 13*z
print -8*w - 16*x - 4*y - 44*z
```

We see exactly what we wanted, namely the right-hand sides of our original equations: 154, 17, -1, and -52.

Likewise, we can just backspace over those four variable settings and provide the settings for any specific solution in our list above. More precisely, in the case of the seventh entry in our above list, we would type

```
w=8-2*pi-4*sqrt(2)
x=pi
y=-3-3*sqrt(2)
z=sqrt(2)
print 20*w + 40*x + 2*y + 86*z
print 4*w + 8*x + 5*y + 31*z
print w + 2*x + 3*y + 13*z
print -8*w - 16*x - 4*y - 44*z
```

As you can see, we get the output that we wanted: 154, 17, -1, and -52.

D.2. Another Example

Let's imagine you are asked to find all solutions to the linear system of equations

$$\begin{aligned} 3w - 3y + 6z &= 51 \\ x + 4y + 5z &= 19 \\ -w + x + 5y + 3z &= 2 \\ 2w + 5x + 18y + 29z &= 129 \end{aligned}$$

We would begin by converting this to a matrix,

$$\left[\begin{array}{ccccc} 3 & 0 & -3 & 6 & 51 \\ 0 & 1 & 4 & 5 & 19 \\ -1 & 1 & 5 & 3 & 2 \\ 2 & 5 & 18 & 29 & 129 \end{array}\right]$$

which has as its RREF, the matrix

$$\left[\begin{array}{ccccc} 1\star & 0 & -1 & 2 & 17 \\ 0 & 1\star & 4 & 5 & 19 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{array}\right]$$

First, we identify the pivots. As you can see, I have marked them with $\star$s. Second, we must classify the variables as "free" or "determined." In this case, since w and x have pivots, thus they are determined variables; likewise, y and z do not have pivots, so they are free variables. Now we write down the uninterpreted literal translation of the RREF matrix into algebra, skipping the all-zero rows

$$\begin{aligned} w - y + 2z &= 17 \\ x + 4y + 5z &= 19 \end{aligned}$$

Finally, we move the free variables (y and z) to the right of the equal sign, and obtain

$$\begin{array}{rcl} w & = & 17 + y - 2z \\ x & = & 19 - 4y - 5z \\ y & \text{is} & \text{free} \\ z & \text{is} & \text{free} \end{array}$$

Perhaps if we were asked to find four distinct solutions, we could write:

(1) If $y = 0$ and $z = 0$ then $w = 17$ and $x = 19$.
(2) If $y = 0$ and $z = 1$ then $w = 15$ and $x = 14$.
(3) If $y = 1$ and $z = 0$ then $w = 18$ and $x = 15$.
(4) If $y = 1$ and $z = 1$ then $w = 16$ and $x = 10$.

A Bit More Terminology

We've used the terms "free variable" and "determined variable," as well as "elementary column" and "degenerate column." The term *degenerate system* indicates[2] any system of linear equations where the solution is not unique—regardless if it has no solution, or if it has infinitely many solutions.

To distinguish among systems of linear equations with infinitely many variables, we will use the concept of "degrees of freedom." The number of degrees of freedom is the number of free variables. Here, both systems which we've seen up to this point have two degrees of freedom. We'll see several of them shortly that have only one degree of freedom. There also exist systems with three or more degrees of freedom (i.e. three free variables), but they seem to be very rare in applications, and therefore they are not emphasized here. Also, a solution where all the variables are determined and no variables are free (a unique solution) is said to have zero degrees of freedom. Normally one does not use the "degrees of freedom" vocabulary for a system without solutions.

The number of degrees of freedom of a system of equations is also called *the right nullity* of the matrix which describes that system of equations. The number of determined variables is called *the rank* of the matrix. The rank of matrix comes up in many places throughout the study of matrix algebra.

Challenge Yourself:

Here are four easy examples which you can use to challenge yourself, to see if you've got this right. The final few equations with the word "free" are given at the end of this appendix. You should also generate 3–4 specific

[2]The word degenerate is—in the modern era—extremely rude. However, at one time it must have been polite, or at least socially acceptable. The word was used in the third quarter of the twentieth century to describe any person who was significantly different from societal norms. Here, a "normal" linear system of equations is one with a unique solution. Any deviation from that expected behavior is termed degenerate.

solutions, and see if they actually work or not. We will provide a specific solution where all the free variables are set to 1.

$$M_1 = \left[\begin{array}{cccc|c} 1 & 2 & -1 & 0 & 1 \\ 3 & 6 & -3 & 2 & 7 \\ 2 & 4 & -2 & -1 & 0 \\ 7 & 14 & -7 & 6 & 19 \end{array}\right] \qquad M_2 = \left[\begin{array}{cccc|c} 1 & 3 & 0 & 0 & 1 \\ 3 & 9 & 2 & 0 & 7 \\ 2 & 6 & -1 & -4 & -32 \\ 7 & 21 & 6 & 5 & 59 \end{array}\right]$$

$$M_3 = \left[\begin{array}{ccc|c} 4 & 20 & -24 & 16 \\ 7 & 35 & -42 & 28 \\ 1 & 5 & -6 & 4 \end{array}\right] \qquad C_2 = \left[\begin{array}{ccc|c} 1 & 2 & 3 & 7 \\ 4 & 5 & 6 & 16 \\ 7 & 8 & 9 & 25 \end{array}\right]$$

By the way, that's the same C_2 that we saw back on Page 29. At that time, we computed that `C2.rref()` equals

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

and then we stopped and said "there are infinitely many solutions." However, now I'd like you to produce equations with free and determined variables as we've done throughout this appendix.

D.3. Exploring Deeper

At this point, the majority of the topic is familiar to you, but the following minor points will give you further insight.

D.3.1. An Interesting Re-Examination of Unique Solutions

Let's briefly return to B from the Section 1.5.3, "Matrices and Sage, Part One" on Page 23. The original matrix and its RREF are

$$B = \left[\begin{array}{cccc|c} -1 & 2 & 1 & -5 & 6 \\ 0 & 0 & -1 & 1 & -5 \\ 1 & 3 & 3 & -1 & 0 \\ -1 & 5 & -1 & 0 & -1 \end{array}\right] \Rightarrow \text{becomes} \Rightarrow \left[\begin{array}{cccc|c} 1\star & 0 & 0 & 0 & -107/7 \\ 0 & 1\star & 0 & 0 & -12/7 \\ 0 & 0 & 1\star & 0 & 54/7 \\ 0 & 0 & 0 & 1\star & 19/7 \end{array}\right]$$

Again, I have marked the pivots with $\star$s. As you can see, all the columns are elementary and none are degenerate. All the variables are determined, and none of them are free. For this reason, we have one and only one solution:

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

D.3.2. Two Equations and Four Unknowns

It would be slightly odd, but someone could ask you to produce solutions to the following two equations.

$$\begin{aligned} 11w + 88x - 11y + 13z &= 47 \\ 17w + 136x - 17y + 19z &= 65 \end{aligned}$$

The reason this looks odd is that, while we have 4 unknowns, we only have 2 variables. It seems as though we have "too few" equations and that there is not enough information given in the problem to solve it. Actually, this intuition is correct. It turns out that there's a theorem—when there are fewer equations than unknowns, either there is no solution or there's infinitely many solutions—you cannot end up with a unique solution. We can't prove that theorem in general here, but we can explore this specific example.

First, it is extremely straightforward to convert this system of equations into a matrix, though one that looks a bit odd since it is a 2×5 matrix.

$$\left[\begin{array}{cccc|c} 11 & 88 & -11 & 13 & 47 \\ 17 & 136 & -17 & 19 & 65 \end{array}\right]$$

We can ask Sage to find the RREF, and then we can identify the pivots. The RREF is given below, and the pivots have been marked with $\star$s.

$$\left[\begin{array}{cccc|c} 1\star & 8 & -1 & 0 & -4 \\ 0 & 0 & 0 & 1\star & 7 \end{array}\right]$$

Now using our standard methods, we finish with:

$$\begin{aligned} w &= -4 - 8x + y \\ x &\text{ is free} \\ y &\text{ is free} \\ z &= 7 \end{aligned}$$

What does this mean? It means that you can pick any x and any y that you want. However, once you've made that choice, the w is determined by the formula $w = -4 - 8x + y$. In contrast to all that change in w, x, and y, the z is very stereotyped and forever locked at $z = 7$. It is interesting to note that we have two degrees of freedom, but simultaneously, there are no rows of zeros in the RREF.

D.3.3. Two Equations and Four Unknowns, but No Solutions:

It is possible that the curious reader might be interested in a system of equations with 2 equations, 4 variables, but no solutions. Here is an obvious example:

$$\begin{aligned} w + x + y + z &= 1 \\ w + x + y + z &= 2 \end{aligned}$$

Clearly, there is no way that $w+x+y+z$ could be both equal to 1 and equal to 2 at the same time.

D.3.4. Four Equations and Three Unknowns

Consider the system of 4 equations and 3 unknowns given below:

$$\begin{aligned} x+2y+3z &= 31 \\ 4x+5y+6z &= 73 \\ 7x+8y-z &= 35 \\ 5x+7y+9z &= 104 \end{aligned}$$

This produces the matrix

$$G = \left[\begin{array}{ccc|c} 1 & 2 & 3 & 31 \\ 4 & 5 & 6 & 73 \\ 7 & 8 & -1 & 35 \\ 5 & 7 & 9 & 104 \end{array}\right]$$

with the RREF

$$\texttt{G.rref} = \left[\begin{array}{ccc|c} 1\star & 0 & 0 & 5 \\ 0 & 1\star & 0 & 1 \\ 0 & 0 & 1\star & 8 \\ 0 & 0 & 0 & 0 \end{array}\right]$$

As you can see, there are three pivots—which I have marked with $\star$s. Therefore, each of the variables is determined, and none of them are free. Finally, we conclude that there is a single solution, with $x=5$, $y=1$, and $z=8$. Note there is one unique solution, despite the presence of a row of all zeros.

D.4. Misconceptions

Now I'd like to briefly address a few misconceptions that students often have about this topic of infinitely many solutions to a linear system of equations.

A Major Misconception:

Frequently, students imagine that if there is a row of zeros, then there will be infinitely many solutions. However, we saw in Section D.3.2, a case where a matrix had no row of zeros—but it had infinitely many solutions. In Section D.3.4, we saw a case where the matrix had a row of zeros, but it had exactly one solution. Therefore, what does a matrix tell you, when one of the rows is all zeros?

If you have the same number of variables as equations, then you can use the rule of thumb from Section 1.5.8 on Page 28. Otherwise, the row of zeros tells you absolutely nothing. Instead, you must identify the pivots, and then determine which variables are free and which are determined—as we've been doing throughout this appendix.

A Less Common Misconception:

Sometimes students will imagine that a column is effective if it is zero everywhere but one entry with a one in it. While this is almost always true, here is a counterexample that shows this belief is false. Consider the matrix

$$Q = \left[\begin{array}{ccc|c} 1\star & 0 & 1 & 3 \\ 0 & 1\star & 0 & 2 \\ 0 & 0 & 0 & 0 \end{array}\right]$$

As you can see, the column for each variable is zero everywhere, except for a single one. A student with this misconception would imagine that each variable is determined, and that none are free. This confused student would say there is one single solution: $x = 3$, $y = 2$, and $z = 0$. However, that's clearly false.

In particular, I draw your attention to the third column, where there is a single 1 and all the entries are zero. That column looks to be elementary, but because that lonesome 1 is not a pivot, the column is not elementary, but is instead degenerate. Here you can see the genuine value of marking the pivots with $\star$s. The pivots in this matrix are Q_{11} and Q_{22}, yet Q_{13} is not a pivot. Clearly, column three has no pivot. Therefore, z is free. We should write

$$\begin{array}{rcl} x & = & 3 - z \\ y & = & 2 \\ z & \text{is} & \text{free} \end{array}$$

to describe the infinite set of solutions.

D.5. Formal Definitions of REF and RREF

For the very curious, the formal definitions of pivot, REF, and RREF are given below.

- An entry of a matrix is a *pivot* if and only if it is the leftmost non-zero entry in its row.
- Therefore, by definition, there are never any non-zero entries to the left of any pivot.
- If a matrix is such that there are no non-zero entries to left of, or below, any pivot then that matrix is said to be in *row-echelon form* or REF.
- If a matrix is such that there are no non-zero entries to left of, below, or above, any pivot then that matrix is said to be in *reduced row-echelon form* or RREF.
- Sometimes, a matrix is such that there are no non-zero entries to the left of, below, above, or even to the right of any pivot.

- If the system is exactly defined, that's called a *permutation* matrix. When solving a system of equations with a permutation matrix, you'll see that the unknowns are just the constants, though probably in a different order. Permutation matrices come up in several branches of linear algebra, both pure and applied.
- If the system is over-defined or under-defined, this is called a *rook* matrix because of some interesting problems related to mathematics and chess, but they also come up in combinatorics problems.

D.6. Alternative Notations

A more sophisticated notation exists for exploring these spaces of infinitely many solutions. If you have already studied vectors, then you might find this section extremely interesting. However, if you are not very comfortable with vectors, you should probably stop reading this appendix now. Alternatively, you can jump to Section D.7, "Geometric Interpretations in 3 Dimensions," on Page 318 if you like.

Consider the first example of this appendix, but writing $x = x$ instead of "x is free," and $z = z$ instead of "z is free." We obtain:

$$\left.\begin{array}{rcl} w &=& 8-2x-4z \\ x &=& x \\ y &=& -3-3z \\ z &=& z \end{array}\right\} \Rightarrow \begin{pmatrix} w \\ x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 8 \\ 0 \\ -3 \\ 0 \end{pmatrix} + x\begin{pmatrix} -2 \\ 1 \\ 0 \\ 0 \end{pmatrix} + z\begin{pmatrix} -4 \\ 0 \\ -3 \\ 1 \end{pmatrix}$$

Now consider the second example of this appendix, but writing $y = y$ instead of "y is free," and $z = z$ instead of "z is free." We obtain:

$$\left.\begin{array}{rcl} w &=& 17+y-2z \\ x &=& 19-4y-5z \\ y &=& y \\ z &=& z \end{array}\right\} \Rightarrow \begin{pmatrix} w \\ x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 17 \\ 19 \\ 0 \\ 0 \end{pmatrix} + y\begin{pmatrix} 1 \\ -4 \\ 1 \\ 0 \end{pmatrix} + z\begin{pmatrix} -2 \\ -5 \\ 0 \\ 1 \end{pmatrix}$$

To make sure you've got it, we've got two more challenges:

5. Convert M_3 from Page 314 into this vector-based notation.
6. Convert C_2 from Page 314 into this vector-based notation.

D.7. Geometric Interpretations in 3 Dimensions

There are some remarkably useful ways to visualize these sorts of problems geometrically. Just as $2x+3y = 6$ represents a line in a plane, $x+2y+3z = 6$ represents a plane floating in space. When you intersect two lines (or solve a system of 2 linear equations in 2 unknowns) then you usually are computing the point where the two lines intersect. However, there are the "minor cases." One minor case is when the two lines are parallel, and thus never

intersect. The other minor case is when both lines are actually the same line, and there is an infinite number of solutions.

When you have a system of three linear equations and three variables, you are intersecting three planes in ordinary space. Usually, the three planes will intersect at a common point, but once again, there are the minor cases. This time there are 4 minor cases.

- The first is that it is possible that the three planes never have a point in common: much like two opposite walls of a rectangular room and the roof of that room. Since the two opposite walls are parallel, they never have a point in common.
- The second is when the three planes have a line in common. Imagine a hardcover book that doesn't have too many pages, partially open, with very stiff pages. The three planes can be any of the very stiff pages, or either of the covers. Furthermore, these planes have the spine of the book (which is a line) in common.
- The third or truly rare case for three variables is when all three planes are actually the same plane, and any point in that plane solves all three equations. However, that does not mean that every point in three-dimensional space is a solution. Rather, a point has to be in the plane to be a solution.
- A fourth (non-sense) case is the following matrix:

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

  where each of the three equations is actually the trivial equation:

$$0x + 0y + 0z = 0$$

  and therefore every point in 3-dimensional space satisfies these "equations." The solution space is our entire 3-dimensional universe.

D.7.1. Visualization of These Cases

Using Sage, each of these pictures can be drawn. However, being 3D images, they can only be seen on your screen where there can be color, your mouse can move the images around to see them from various angles, and the planes can be made transparent. However, I cannot put meaningful pictures of that sort on the printed page of a black-and-white book.

Instead, I have made an applet for you. This applet, or interactive webpage, is titled "Visualizing Infinitely Many Solutions in a Linear System of 3 Equations and 3 Unknowns" and can be found on my webpage `www.gregorybard.com` after clicking on "Interactive Web Pages for Teaching Math." Three examples are given there. For the "zero degrees of freedom" case, we use the matrix A from Page 19; for the "one degree of freedom"

case, we use the matrix C_2 which appeared on Page 29 and Page 314; for the "two degrees of freedom" case, we use the matrix M_3 from Page 314.

D.7.2. Geometric Interpretations in Higher Dimensions

Problems with four or more variables require the notions of high-dimensional spaces and hyperplanes. For example, with 2 equations in 4 unknowns in the RREF, you often have the solutions turning out to be a plane floating in 4-dimensional space. With 3 equations in 4 unknowns, the solution is often a 3-dimensional hyperplane in that 4-dimensional space. The hyperplane has the same geometry as the universe that we live in. This all sounds very complicated, but it can get worse, actually.

With 5 unknowns and 2 equations in the RREF, the solutions often are a subspace of 3-dimensions floating in a 5-dimensional space. (That's three degrees of freedom, by the way.) One way to imagine this is a universe larger and more complicated than our own, but one tiny subspace of that universe is our entire "everyday universe." That tiny subspace is the solution set.

After much coursework, mathematicians can learn to understand these ideas formally, in the sense of algebra. However, only very few people can actually *visualize* these objects. The vast majority of us, including myself, cannot—for many and myself, visualization ends at 3-dimensions. Therefore, we will not explore the geometrical point of view in high dimensions any further.

D.8. Dummy-Variable Notation

Many textbooks insist that the free variables be rewritten as s and t when there are two degrees of freedom, and just t when there is one degree of freedom. This is an extremely odd choice, as it seems to blur the line between free variables and determined variables. (Just for completeness, when there are three degrees of freedom, the free variables would be changed to r, s, and t.)

If we felt compelled to follow that notation then we would have a very simple (but very unneeded) extra step when processing the first example of this appendix. Namely, we would make the following change:

$$\left.\begin{array}{rcl} w &=& 8-2x-4z \\ x &=& x \\ y &=& -3-3z \\ z &=& z \end{array}\right\} \Rightarrow \left\{\begin{array}{rcl} w &=& 8-2s-4t \\ x &=& s \\ y &=& -3-3t \\ z &=& t \end{array}\right.$$

and similarly, for the second example of this appendix,

$$\left.\begin{array}{rcl} w &=& 17+y-2z \\ x &=& 19-4y-5z \\ y &=& y \\ z &=& z \end{array}\right\} \Rightarrow \left\{\begin{array}{rcl} w &=& 17+s-2t \\ x &=& 19-4s-5t \\ y &=& s \\ z &=& t \end{array}\right.$$

At first glance, this change is an extra step, yet appears to accomplish nothing. However, this notation is not totally without merits. For example, a system with one degree of freedom has a t in it, which is always the free variable. Likewise, with two degrees of freedom, there will be an s and t. So there is no doubt as to which variables are free and how many of them are free. However, similar things can be said for the notation that I used throughout the appendix, by using the word "free." Moreover, it seems a pity to turn a 4-variable problem into a 6-variable problem—4-variable problems are hard enough as it is. Last but not least, t is often confused with $+$, and similarly s is often confused with 5. All things considered, I think it is best to avoid introducing new variables.

However, if your instructor forces you to use t, then note that a capital T is easier to distinguish from a $+$ sign than a lowercase t would be.

D.9. Solutions to Challenges

(1) For M_1 the solution is

$$\begin{array}{rcl} w & = & 1-2x+y \\ x & \text{is} & \text{free} \\ y & \text{is} & \text{free} \\ z & = & 2 \end{array}$$

A sample solution would be $w=0$, $x=1$, $y=1$, $z=2$.

(2) For M_2 the solution is

$$\begin{array}{rcl} w & = & 1-3x \\ x & \text{is} & \text{free} \\ y & = & 2 \\ z & = & 8 \end{array}$$

A sample solution would be $w=-2$, $x=1$, $y=2$, $z=8$.

(3) For M_3 the solution is

$$\begin{array}{rcl} x & = & 4-5y+6z \\ y & \text{is} & \text{free} \\ z & \text{is} & \text{free} \end{array}$$

A sample solution would be $x=5$, $y=1$, $z=1$.

(4) For C_2 the solution is

$$\begin{array}{rcl} x & = & -1+z \\ y & = & 4-2z \\ z & \text{is} & \text{free} \end{array}$$

A sample solution would be $x=1$, $y=-2$, $z=1$.

(5) Here we transform the solution for M_3 into vector notation.

$$\left.\begin{array}{rcl} x &=& 4-5y+6z \\ y &=& y \\ z &=& z \end{array}\right\} \Rightarrow \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 4 \\ 0 \\ 0 \end{pmatrix} + y \begin{pmatrix} -5 \\ 1 \\ 0 \end{pmatrix} + z \begin{pmatrix} 6 \\ 0 \\ 1 \end{pmatrix}$$

(6) Here we transform the solution for C_2 into vector notation.

$$\left.\begin{array}{rcl} x &=& -1+z \\ y &=& 4-2z \\ z &=& z \end{array}\right\} \Rightarrow \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} + z \begin{pmatrix} 1 \\ -2 \\ 1 \end{pmatrix}$$

Appendix E

Installing Sage on your Personal Computer

My customary response when people ask me about installing Sage on their personal computer is "Are you sure that you want to do this?"

Reasons Against a Local Installation:

Part of the genius of Sage is that it brings cutting-edge mathematical software to anyone who has access to the internet. From any web-browser, you can use the Sage Cell Server for small computations, and for large computations, you can use SageMathCloud.

Keeping your work on SageMathCloud has many advantages over a local installation. For example, if your laptop is lost or stolen, you do not lose your files. If you need to do work while traveling, you can use any computer with an internet connection. It is very easy to share your projects with others in SageMathCloud, by giving them electronic access. If your computation has to run for a few hours, you don't have to keep your laptop "tied up" by the computation if the computation is being done on SageMathCloud. All the software packages will always be up to date, without any extra effort on the part of the user. Personally, I have never had a local installation of Sage.

In the classroom, this is particularly an important point to dwell on. If an instructor asks a group of 40 students to do a local install, the instructor will have to deal with their various operating systems, various machines, and student inexperience. Many students might be rather computer literate but probably the worst 2–3 of those 40 would be significantly less computer literate. As computers become easier and easier to use, the skill level of the average user tends to be decreasing. For example, the concept of the directory tree and subdirectories is now one that students find confusing and difficult to understand at first. Students are often extremely intimidated and hostile to the "command line," or are at best unfamiliar with it.

Often, students who have a grudge against an instructor (for having high standards or difficult examinations) find themselves unable to complain to

department chairs and Deans on that basis, and so they can use the difficulty of using mandatory computer software as a basis for complaint. For example, "But, Dean XYZ, you don't understand, I couldn't even get the computer program to install!"

Reasons in Favor of a Local Installation:

Upon further thought, there are also some advantages to performing a local installation. While broadband internet access is ubiquitous in most of North America, this is less the case in rural China, sub-Saharan Africa, and (I am told) parts of rural Wisconsin and other sparsely populated US states. One enormous disadvantage of both the Sage Cell Server and SageMathCloud is that they require internet access. Several members of the Sage community are involved with the African Institute for Mathematical Sciences, a community attracted to Sage due to its absence of cost. Furthermore, there are some parents in the USA who refuse to permit their children to have access to the internet at all, on a religious basis. This might be a factor for some high school teachers. One of my former undergraduate research assistants grew up in such a household.

On the positive side, while installing Sage on your machine takes a lot of time and effort, the result will be that it runs much faster. A lot of the delay in using Sage is the transit time between your computer and the servers. For those with internet connections that are slow, this is an important factor.

Instructions for the Mac:

If you are using a Mac, and have OS X, then you should go to this URL:

`http://www.sagemath.org/download-mac.html`

Further instructions are found at:

`http://www.sagemath.org/mirror/osx/README.txt`

The Sage LiveCD:

This is a bootable CD from which you can run Sage directly. It can be found at this URL:

`http://www.sagemath.org/download-livecd.html`

Instructions for Unix and Linux Users:

You can find binaries for a Unix or Linux install at this URL:

`http://www.sagemath.org/download-linux.html`

However, you might also want to build the entire system from scratch, if you are an experienced software developer. That's part of the fun of open-source software, after all. Because Sage is a large and complex system, this might take more time than you might first anticipate. Those bold and experienced users should visit this URL:

`http://www.sagemath.org/download-source.html`

Instructions for Microsoft Windows:

The download for Windows users can be found at the following URL:

`http://www.sagemath.org/download-windows.html`

However, I strongly recommend the instructions found at:

`http://wiki.sagemath.org/SageAppliance`

As it comes to pass, you'll be installing VirtualBox, which is a virtual machine that will encapsulate Sage. This is a highly tested and safe process. I am told that there is almost no performance lag compared to Linux and Mac installations.

Appendix F

Index of Commands by Name and by Section

Here we have a comprehensive index of all the commands in this book. First, we present them sorted thematically (by section). Second, we present them sorted alphabetically, by name. Many thanks to Thomas Suiter, who compiled this index.

Section	Command	Description
1.2	sqrt	Computes the square root
1.2	N	Returns a decimal approximation
1.2	numerical_approx	Returns a real number, if possible. Longhand for the command "n"
1.2	exp	An abbreviated notation when dealing with exponentials, namely e^x
1.2	log	Computes any logarithm, but takes the natural logarithm by default
1.3	sin	Calls the trigonometric function "sine" in radians
1.3	cos	Calls the trigonometric function "cosine" in radians
1.3	tan	Calls the trigonometric function "tangent" in radians
1.3	csc	Calls the trigonometric function "cosecant" in radians
1.3	sec	Calls the trigonometric function "secant" in radians
1.3	cot	Calls the trigonometric function "cotangent" in radians
1.3	arcsin	Calls the inverse trigonometric function of sine in radians
1.3	arccos	Calls the inverse trigonometric function of cosine in radians
1.3	arctan	Calls the inverse trigonometric function of tangent in radians
1.3	arccsc	Calls the inverse trigonometric function of cosecant in radians
1.3	arcsec	Calls the inverse trigonometric function of secant in radians
1.3	arccot	Calls the inverse trigonometric function of cotangent in radians
1.3	asin	Shorthand notation for the "arcsin" function
1.3	acos	Shorthand notation for the "arccos" function
1.3	atan	Shorthand notation for the "arctan" function
1.3	acsc	Shorthand notation for the "arccsc" function
1.3	asec	Shorthand notation for the "arcsec" function
1.3	acot	Shorthand notation for the "arccot" function
1.4	plot	Plots a given single-variable function in the coordinate plane
1.4	abs	Returns the absolute value of a real number

1.4	point	Used to identify/mark a single point in a plot, by making a large dot there
1.4.2	show	Used when superimposing a large number of plots on top of one another
1.5.3	matrix	Used to define a matrix, given the number of rows, number of columns, and its entries
1.5.3	rref	Computes the "reduced row echelon form" of the given matrix
1.5.4	print	Used to print information to the screen for the user
1.7	expand	Gives the expanded form of a polynomial (or rational function) without parentheses
1.7	factor	Gives a factored version of a polynomial or an integer
1.7	gcd	Computes the "greatest common divisor" of two polynomials or numbers
1.7	divisors	Returns a list of all the positive integers that divide into the original integer
1.8.1	solve	Returns a symbolic solution to an equation or system of equations
1.8.1	var	Declares a new variable to represent some unknown quantity
1.9	find_root	A numerical approximation of $f(x) = 0$
1.10	search_doc	Searches Sage for all of the documentation strings regarding the specified command
1.11	diff	Takes the derivative by specifying the function in terms predefined variable(s)
1.11	derivative	A synonym for diff
1.12	integral	Takes the integral by specifying the function in terms predefined variable(s)
1.12	integrate	A synonym for integral
1.12	numerical_integral	Calculates the integral numerically, outputting the best guess first and the uncertainty second
1.12	partial_fraction	Computes the partial fractions of a complicated rational function
1.12	erf	Standing for "Error Function", sometimes called the Gaussian or Normal Distribution
1.12	assume	Allows user to declare an assumption

2.4.3	lcm	Finds the least common multiple of two numbers (See also 4.6.1)
2.6.1	factorial	Takes the factorial of any non-negative integer
2.6.2	mod	Take modular reduction only of the nearest number (See also 4.6.6)
3.1.1	axes_labels	Labels the axes directly on a graph
3.1.3	arrow	Draws an arrow directly on a graph
3.1.3	text	Used to add words onto a graph
3.3	polar_plot	Used to graph a function with polar coordinates
3.4	implicit_plot	Takes a function with two variables and plots the graph $f(x, y) = 0$
3.5	contour_plot	Takes a function with two variables and plots contour curves (level sets) of that function
3.5.1	sum	Used to take the summation of an expression (properly shown in 4.19)
3.6	parametric_plot	Takes two or three functions and plots a parametric curve
3.7	plot_vector_field	Takes two functions with two different variables and plots vector arrows
3.7.1	plot3d	Creates a 3D plot of a two-variable function
3.7.2	vector	Creates a single mathematical vector
3.7.2	norm	Computes the norm (or length) of a vector
3.8	plot_loglog	Plots a given single-variable function in the coordinate plane with logarithmic scales
3.8	scatter_plot	Plots all of the predefined data points onto a graph (see Section 4.9 also)
3.9.2	nth_root	A method of finding the nth real root (see Section 3.9.2)
3.9.2	sign	Returns either a negative or positive 1 depending on if x is negative or positive, respectively
3.9.2	real_nth_root	[Currently proposed but does not yet exist] Computes the real nth root of a number

4.4.2	identity_matrix	Creates the identity matrix of size n
4.4.3	solve_right	Solves the matrix-vector problem $A\vec{x} = \vec{b}$ where $\vec{x}$ is unknown.
4.4.3	solve_left	Solves the matrix-vector problem $\vec{x}A = \vec{b}$ where $\vec{x}$ is unknown.
4.4.3	augment	Adds onto a predefined matrix
4.4.3	echelon_form	A half-completed version of RREF
4.4.4	inverse	Computes the inverse of a predefined matrix
4.4.5	left_kernel	Computes the set of vectors n such that $\vec{n}A = \vec{0}$, for a matrix A
4.4.5	right_kernal	Computes the set of vectors n such that $A\vec{n} = \vec{0}$, for a matrix A
4.4.5	right_nullity	Computes the dimension of the set of vectors n such that $A\vec{n} = \vec{0}$, for a matrix A.
4.4.5	left_nullity	Computes the dimension of the set of vectors n such that $\vec{n}A = \vec{0}$, for a matrix A.
4.4.6	det	Calculates the determinant of a predefined matrix
4.5	dot_product	Calculates the dot product of a set of vectors
4.5	cross_product	Calculates the cross product of a set of vectors
4.6	next_prime	Finds the next prime number greater than n
4.6	is_prime	Test the number for primality
4.6	prime_range	Returns all of the prime numbers between the desired range
4.6.1	lcm	Finds the least common multiple of two numbers (See also 2.4.3)
4.6.2	nth_prime	Used to find the nth prime number
4.6.3	euler_phi	Returns how many positive integers (from 1 to n) are coprime to n.
4.6.4	sigma	Returns the sum of the positive integers which divide the given positive integer
4.6.4	len	Returns the length of any list
4.6.6	mod	Take modular reduction only of the nearest number (See also 2.6.2)
4.7	min	Returns the smallest number in the list

4.7	max	Returns the largest number in the list
4.7.1	floor	Sometimes called The Greatest Integer Function, this function rounds x down
4.7.1	ceil	Sometimes called The Least Integer Function, this function rounds x up
4.7.2	binomial	Computes the binomial coefficient or "choose" function for combinations
4.7.2	list	Returns a list of what is being asked
4.7.2	Permutations	The input is a list of items; the output is a list of all possible permutations of that list.
4.7.3	sinh	Calls the hyperbolic trigonometric function of sine in radians
4.7.4	cosh	Calls the hyperbolic trigonometric function of cosine in radians
4.7.5	tanh	Calls the hyperbolic trigonometric function of tangent in radians
4.7.5	coth	Calls the hyperbolic trigonometric function of cotangent in radians
4.7.5	sech	Calls the hyperbolic trigonometric function of secant in radians
4.7.5	csch	Calls the hyperbolic trigonometric function of cosecant in radians
4.7.5	arcsinh	Calls the inverse hyperbolic trigonometric function of sine in radians
4.7.5	arccosh	Calls the inverse hyperbolic trigonometric function of cosine in radians
4.7.5	arctanh	Calls the inverse hyperbolic trigonometric function of tangent in radians
4.7.5	arccoth	Calls the inverse hyperbolic trigonometric function of cotangent in radians
4.7.5	arcsech	Calls the inverse hyperbolic trigonometric function of secant in radians
4.7.5	arccsch	Calls the inverse hyperbolic trigonometric function of cosecant in radians
4.7.5	asinh	Shorthand notation for the "arcsinh" function
4.7.5	acosh	Shorthand notation for the "arccosh" function
4.7.5	atanh	Shorthand notation for the "arctanh" function
4.7.5	acoth	Shorthand notation for the "arccoth" function
4.7.5	asech	Shorthand notation for the "arcsech" function
4.7.5	acsch	Shorthand notation for the "arccsch" function
4.8	limit	Used to take the limit of a function

4.9	scatter_plot	Plots all of the predefined data points onto a graph (see Section 4.9 also)
4.10	find_fit	Finds the best fit line for a list of data points
4.11	hex	Converts the number into hexadecimal
4.11	bin	Converts the number into binary
4.11	oct	Converts the number into octal
4.11	int	Rounds down to the nearest integer
4.12	sudoku	Finds a solution to the sudoku puzzle
4.13	timeit	Times how long it takes Sage to calculate the solution
4.15	latex	Returns the solution written in LaTeX
4.16.2	eigenvalues	Calculates the eigenvalues corresponding to a linear system of equations
4.16.2	eigenvectors_right	Given A, returns a vector $\vec{v}$ such that $A\vec{v} = k\vec{v}$ for some scalar constant k.
4.16.2	eigenvectors_left	Given A, returns a vector $\vec{v}$ such that $\vec{v}A = k\vec{v}$ for some scalar constant k.
4.16.2	charpoly	Calculates the characteristic polynomial
4.16.2	minpoly	Calculates the minimum polynomial
4.16.3	as_sum_of_permutations	This function is one of 14 built-in matrix factorizations.
4.16.3	cholesky	This function is one of 14 built-in matrix factorizations.
4.16.3	frobenius	This function is one of 14 built-in matrix factorizations.
4.16.3	gram_schmidt	This function is one of 14 built-in matrix factorizations.
4.16.3	hessenberg_form	This function is one of 14 built-in matrix factorizations.
4.16.3	hermite_form	This function is one of 14 built-in matrix factorizations.
4.16.3	jordan_form	This function is one of 14 built-in matrix factorizations.
4.16.3	LU	This function is one of 14 built-in matrix factorizations.
4.16.3	QR	This function is one of 14 built-in matrix factorizations.
4.16.3	rational_form	This function is one of 14 built-in matrix factorizations.
4.16.3	smith_form	This function is one of 14 built-in matrix factorizations.
4.16.3	symplectic_form	This function is one of 14 built-in matrix factorizations.

4.16.3	weak_popov_form	This function is one of 14 built-in matrix factorizations.
4.16.3	zigzag_form	This function is one of 14 built-in matrix factorizations.
4.16.4	transpose	Finds the transpose of a particular matrix
4.17.1	taylor	Calculates the Taylor Polynomial approximation of a function
4.18.1	minimize	Finds the minima of a function
4.19	sum	Used to take the summation of an expression (first shown in 3.5.1)
4.20	continued_fraction	Computes the continued fraction expansion of a real number
4.21.1	MixedIntegerLinearProgram	Used to initialize either a linear program or mixed integer linear program
4.21.1	new_variable	Used to create an infinite set of variables for linear programming
4.21.1	add_constraint	Used to add the constraints to a linear program
4.21.1	set_objective	Used to set the objective to a linear program
4.21.1	get_values	Used to obtain the values of variables in a linear program
4.21.2	remove_constraint	Used to remove a constraint from a linear program
4.22.1	function	Declares a new unknown function
4.22.1	desolve	Used when solving for a first or second order linear ordinary differential equation
4.22.3	plot_slope_field	Generates a slope field plot of the desired function
4.23.1	laplace	Calculates the Laplace transform
4.23.3	inverse_laplace	Calculates the inverse Laplace transform
5.1.1	for	Used to create a 'for loop' statement
5.1.1	range	Used as a shortcut for making a list of numbers
5.2.1	def	Used to define a new subroutine
5.3.6	return	Returns what is being asked
5.4.1	if	Used for creating 'if-then' statements
5.5.1	raise	Used to cause an exception when dealing with errors
5.5.1	RuntimeError	Used to specify if there is an error when using the subroutine

5.5.2	else	Used in conjunction with the 'if-then' statement, it handles when the 'if' statement is false
5.6.1	while	Used when creating a 'while loop' statement
5.6.3	break	Exits the current loop statement
5.7.1	append	Adds a new item to the end of the list
5.7.3	in	Returns true or false, to test if an element is in a given set, or not.
5.7.3	remove	Removes an item from the list
5.7.3	prod	Multiplies all of the elements in the list
5.7.3	shuffle	Randomizes all of the elements in the list
5.7.5	or	Compares two logic statements to check if either-or statements are true
5.7.5	and	Compares two logic statements to check if both statements are true
5.7.10	rhs	Given an equation, this returns the "right hand side" of that equation as an expression
5.7.10	lhs	Given an equation, this returns the "left hand side" of that equation as an expression
6.1	@interact	Tells Sage you want an interact (interactive webpage) created
6.1	slider	Creates a slider for a subroutine's input in an interactive webpage
6.5	selector	Creates a small pull-down menu in an interactive webpage, that a user can choose an item from
C	implicit_multiplication	Allows you to use `5x+6y` in place of the usual `5*x + 6*y`

Section	Command	Description
6.1	@interact	Tells Sage you want an interact (interactive webpage) created
1.4	abs	Returns the absolute value of a real number
1.3	acos	Shorthand notation for the “arccos” function
4.7.5	acosh	Shorthand notation for the “arccosh” function
1.3	acot	Shorthand notation for the “arccot” function
4.7.5	acoth	Shorthand notation for the “arccoth” function
1.3	acsc	Shorthand notation for the “arccsc” function
4.7.5	acsch	Shorthand notation for the “arccsch” function
4.21.1	add_constraint	Used to add the constraints to a linear program
5.7.5	and	Compares two logic statements to check if both statements are true
5.7.1	append	Adds a new item to the end of the list
1.3	arccos	Calls the inverse trigonometric function of cosine in radians
4.7.5	arccosh	Calls the inverse hyperbolic trigonometric function of cosine in radians
1.3	arccot	Calls the inverse trigonometric function of cotangent in radians
4.7.5	arccoth	Calls the inverse hyperbolic trigonometric function of cotangent in radians
1.3	arccsc	Calls the inverse trigonometric function of cosecant in radians
4.7.5	arccsch	Calls the inverse hyperbolic trigonometric function of cosecant in radians
1.3	arcsec	Calls the inverse trigonometric function of secant in radians
4.7.5	arcsech	Calls the inverse hyperbolic trigonometric function of secant in radians
1.3	arcsin	Calls the inverse trigonometric function of sine in radians
4.7.5	arcsinh	Calls the inverse hyperbolic trigonometric function of sine in radians
1.3	arctan	Calls the inverse trigonometric function of tangent in radians
4.7.5	arctanh	Calls the inverse hyperbolic trigonometric function of tangent in radians
3.1.3	arrow	Draws an arrow directly on a graph
4.16.3	as_sum_of_permutations	This function is one of 14 built-in matrix factorizations.

1.3	asec	Shorthand notation for the “arcsec” function
4.7.5	asech	Shorthand notation for the “arcsech” function
1.3	asin	Shorthand notation for the “arcsin” function
4.7.5	asinh	Shorthand notation for the “arcsinh” function
1.12	assume	Allows user to declare an assumption
1.3	atan	Shorthand notation for the “arctan” function
4.7.5	atanh	Shorthand notation for the “arctanh” function
4.4.3	augment	Adds onto a predefined matrix
3.1.1	axes_labels	Labels the axes directly on a graph
4.11	bin	Converts the number into binary
4.7.2	binomial	Computes the binomial coefficient or “choose” function for combinations
5.6.3	break	Exits the current loop statement
4.7.1	ceil	Sometimes called The Least Integer Function, this function rounds x up
4.16.2	charpoly	Calculates the characteristic polynomial
4.16.3	cholesky	This function is one of 14 built-in matrix factorizations.
4.20	continued_fraction	Computes the continued fraction expansion of a real number.
3.5	contour_plot	Takes a function with two variables and plots contour curves (level sets) of that function
1.3	cos	Calls the trigonometric function “cosine” in radians
4.7.4	cosh	Calls the hyperbolic trigonometric function of cosine in radians
1.3	cot	Calls the trigonometric function “cotangent” in radians
4.7.5	coth	Calls the hyperbolic trigonometric function of cotangent in radians
4.5	cross_product	Calculates the cross product of a set of vectors
1.3	csc	Calls the trigonometric function “cosecant” in radians
4.7.5	csch	Calls the hyperbolic trigonometric function of cosecant in radians
5.2.1	def	Used to define a new subroutine
1.11	derivative	A synonym for diff

4.22.1	desolve	Used when solving for a first or second order linear ordinary differential equation
4.4.6	det	Calculates the determinant of a predefined matrix
1.11	diff	Takes the derivative by specifying the function in terms predefined variable(s)
1.7	divisors	Returns a list of all the positive integers that divide into the original integer
4.5	dot_product	Calculates the dot product of a set of vectors
4.4.3	echelon_form	A half-completed version of RREF
4.16.2	eigenvalues	Calculates the eigenvalues corresponding to a linear system of equations
4.16.2	eigenvectors_left	Given A, returns a vector $\vec{v}$ such that $\vec{v}A = k\vec{v}$ for some scalar constant k.
4.16.2	eigenvectors_right	Given A, returns a vector $\vec{v}$ such that $A\vec{v} = k\vec{v}$ for some scalar constant k.
5.5.2	else	Used in conjunction with the 'if-then' statement, it handles when the 'if' statement is false
1.12	erf	Standing for "Error Function", sometimes called the Gaussian or Normal Distribution
4.6.3	euler_phi	Returns how many positive integers (from 1 to n) are coprime to n.
1.2	exp	An abbreviated notation when dealing with exponentials, namely e^x
1.7	expand	Gives the expanded form of a polynomial (or rational function) without parentheses
1.7	factor	Gives a factored version of a polynomial or an integer
2.6.1	factorial	Takes the factorial of any non-negative integer
4.10	find_fit	Finds the best fit line for a list of data points
1.9	find_root	A numerical approximation of $f(x) = 0$
4.7.1	floor	Sometimes called The Greatest Integer Function, this function rounds x down
5.1.1	for	Used to create a 'for loop' statement
4.16.3	frobenius	This function is one of 14 built-in matrix factorizations.

4.22.1	function	Declares a new unknown function
1.7	gcd	Computes the "greatest common divisor" of two polynomials or numbers
4.21.1	get_values	Used to obtain the values of variables in a linear program
4.16.3	gram_schmidt	This function is one of 14 built-in matrix factorizations.
4.16.3	hermite_form	This function is one of 14 built-in matrix factorizations.
4.16.3	hessenberg_form	This function is one of 14 built-in matrix factorizations.
4.11	hex	Converts the number into hexadecimal
4.4.2	identity_matrix	Creates the identity matrix of size n
5.4.1	if	Used for creating 'if-then' statements
C	implicit_multiplication	Allows you to use `5x+6y` in place of the usual `5*x + 6*y`
3.4	implicit_plot	Takes a function with two variables and plots the graph $f(x, y) = 0$
5.7.3	in	Returns true or false, to test if an element is in a given set, or not.
4.11	int	Rounds down to the nearest integer
1.12	integral	Takes the integral by specifying the function in terms predefined variable(s)
1.12	integrate	A synonym for integral
4.4.4	inverse	Computes the inverse of a predefined matrix
4.23.3	inverse_laplace	Calculates the inverse Laplace transform
4.6	is_prime	Test the number for primality
4.16.3	jordan_form	This function is one of 14 built-in matrix factorizations.
4.23.1	laplace	Calculates the Laplace transform
4.15	latex	Returns the solution written in LaTeX
2.4.3	lcm	Finds the least common multiple of two numbers (See also 4.6.1)
4.6.1	lcm	Finds the least common multiple of two numbers (See also 2.4.3)
4.4.5	left_kernel	Computes the set of vectors n such that $\vec{n}A = \vec{0}$, for a matrix A
4.4.5	left_nullity	Computes the dimension of the set of vectors n such that $\vec{n}A = \vec{0}$, for a matrix A.
4.6.4	len	Returns the length of any list

5.7.10	lhs	Given an equation, this returns the "left hand side" of that equation as an expression
4.8	limit	Used to take the limit of a function
4.7.2	list	Returns a list of what is being asked
1.2	log	Computes any logarithm, but takes the natural logarithm by default
4.16.3	LU	This function is one of 14 built-in matrix factorizations.
1.5.3	matrix	Used to define a matrix, given the number of rows, number of columns, and its entries
4.7	max	Returns the largest number in the list
4.7	min	Returns the smallest number in the list
4.18.1	minimize	Finds the minima of a function
4.16.2	minpoly	Calculates the minimum polynomial
4.21.1	MixedIntegerLinearProgram	Used to initialize either a linear program or mixed integer linear program
2.6.2	mod	Take modular reduction only of the nearest number (See also 4.6.6)
4.6.6	mod	Take modular reduction only of the nearest number (See also 2.6.2)
1.2	N	Returns a decimal approximation
4.21.1	new_variable	Used to create an infinite set of variables in a linear program
4.6	next_prime	Finds the next prime number greater than n
3.7.2	norm	Computes the norm (or length) of a vector
4.6.2	nth_prime	Used to find the nth prime number
3.9.2	nth_root	A method of finding the nth real root (see Section 3.9.2)
1.2	numerical_approx	Returns a real number, if possible. Longhand for the command "n"
1.12	numerical_integral	Calculates the integral numerically, outputting the best guess first and the uncertainty second
4.11	oct	Converts the number into octal
5.7.5	or	Compares two logic statements to check if either-or statements are true
3.6	parametric_plot	Takes two or three functions and plots a parametric curve

1.12	partial_fraction	Computes the partial fractions of a complicated rational function
4.7.2	Permutations	The input is a list of items; the output is a list of all possible permutations of that list.
1.4	plot	Plots a given single-variable function in the coordinate plane
3.8	plot_loglog	Plots a given single-variable function in the coordinate plane with logarithmic scales
4.22.3	plot_slope_field	Generates a slope field plot of the desired function
3.7	plot_vector_field	Takes two functions with two different variables and plots vector arrows
3.7.1	plot3d	Creates a 3D plot of a two-variable function
1.4	point	Used to identify/mark a single point in a plot, by making a large dot there
3.3	polar_plot	Used to graph a function with polar coordinates
4.6	prime_range	Returns all of the prime numbers between the desired range
1.5.4	print	Used to print information to the screen for the user
5.7.3	prod	Multiplies all of the elements in the list
4.16.3	QR	This function is one of 14 built-in matrix factorizations.
5.5.1	raise	Used to cause an exception when dealing with errors
5.1.1	range	Used as a shortcut for making a list of numbers
4.16.3	rational_form	This function is one of 14 built-in matrix factorizations.
3.9.2	real_nth_root	[Currently proposed but does not yet exist] Computes the real nth root of a number
5.7.3	remove	Removes an item from the list
4.21.2	remove_constraint	Used to remove a constraint from a linear program
5.3.6	return	Returns what is being asked
5.7.10	rhs	Given an equation, this returns the "right hand side" of that equation as an expression
4.4.5	right_kernal	Computes the set of vectors n such that $A\vec{n} = \vec{0}$, for a matrix A

4.4.5	right_nullity	Computes the dimension of the set of vectors n such that $A\vec{n} = \vec{0}$, for a matrix A.
1.5.3	rref	Computes the "reduced row echelon form" of the given matrix
5.5.1	RuntimeError	Used to specify if there is an error when using the subroutine
3.8	scatter_plot	Plots all of the predefined data points onto a graph (see Section 4.9 also)
4.9	scatter_plot	Plots all of the predefined data points onto a graph (see Section 4.9 also)
1.10	search_doc	Searches Sage for all of the documentation strings regarding the specified command
1.3	sec	Calls the trigonometric function "secant" in radians
4.7.5	sech	Calls the hyperbolic trigonometric function of secant in radians
6.5	selector	Creates a small pull-down menu in an interactive webpage, that a user can choose an item from
4.21.1	set_objective	Used to set the objective to a linear program
1.4.2	show	Used when superimposing a large number of plots on top of one another
5.7.3	shuffle	Randomizes all of the elements in the list
4.6.4	sigma	Returns the sum of the positive integers which divide the given positive integer
3.9.2	sign	Returns either a negative or positive 1 depending on if x is negative or positive, respectively
1.3	sin	Calls the trigonometric function "sine" in radians
4.7.3	sinh	Calls the hyperbolic trigonometric function of sine in radians
6.1	slider	Creates a slider for a subroutine's input in an interactive webpage
4.16.3	smith_form	This function is one of 14 built-in matrix factorizations.
1.8.1	solve	Returns a symbolic solution to an equation or system of equations
4.4.3	solve_left	Solves the matrix-vector problem $\vec{x}A = \vec{b}$ where $\vec{x}$ is unknown.
4.4.3	solve_right	Solves the matrix-vector problem $A\vec{x} = \vec{b}$ where $\vec{x}$ is unknown.
1.2	sqrt	Computes the square root

4.12	sudoku	Finds a solution to the sudoku puzzle
4.19	sum	Used to take the summation of an expression (first shown in 3.5.1)
3.5.1	sum	Used to take the summation of an expression (properly shown in 4.19)
4.16.3	symplectic_form	This function is one of 14 built-in matrix factorizations.
1.3	tan	Calls the trigonometric function “tangent” in radians
4.7.5	tanh	Calls the hyperbolic trigonometric function of tangent in radians
4.17.1	taylor	Calculates the Taylor Polynomial approximation of a function
3.1.3	text	Used to add words onto a graph
4.13	timeit	Times how long it takes Sage to calculate the solution
4.16.4	transpose	Finds the transpose of a particular matrix
1.8.1	var	Declares a new variable to represent some unknown quantity
3.7.2	vector	Creates a single mathematical vector
4.16.3	weak_popov_form	This function is one of 14 built-in matrix factorizations.
5.6.1	while	Used when creating a ’while loop’ statement
4.16.3	zigzag_form	This function is one of 14 built-in matrix factorizations.